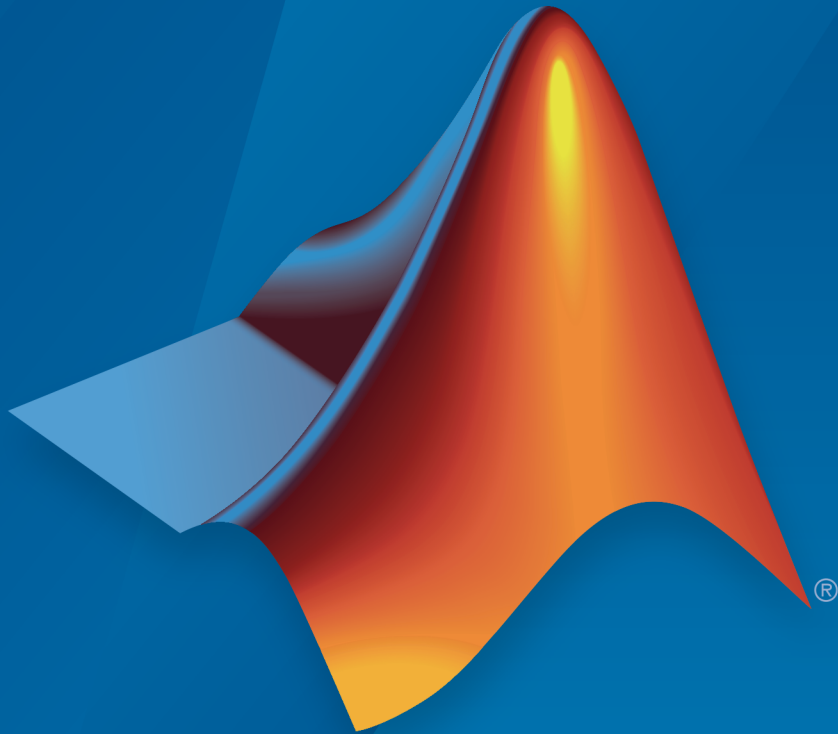


Simulink® Coder™

User's Guide



MATLAB® & SIMULINK®

R2016b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Simulink[®] Coder[™] User's Guide

© COPYRIGHT 2011–2016 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

April 2011	Online only	New for Version 8.0 (Release 2011a)
September 2011	Online only	Revised for Version 8.1 (Release 2011b)
March 2012	Online only	Revised for Version 8.2 (Release 2012a)
September 2012	Online only	Revised for Version 8.3 (Release 2012b)
March 2013	Online only	Revised for Version 8.4 (Release 2013a)
September 2013	Online only	Revised for Version 8.5 (Release 2013b)
March 2014	Online only	Revised for Version 8.6 (Release 2014a)
October 2014	Online only	Revised for Version 8.7 (Release 2014b)
March 2015	Online only	Revised for Version 8.8 (Release 2015a)
September 2015	Online only	Revised for Version 8.9 (Release 2015b)
October 2015	Online only	Rereleased for Version 8.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 8.10 (Release 2016a)
September 2016	Online only	Revised for Version 8.11 (Release 2016b)

Check Bug Reports for Issues and Fixes

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

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

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

Model Architecture and Design

Modeling Environment for Simulink Coder

1

Design Models for Rapid Prototyping Deployment	1-2
Application Algorithms and Run-Time Environments . .	1-2
Software Execution Framework for Generated Code . . .	1-3
Map Embedded System Architecture to Simulink	
Modeling Environment	1-5
Model Templates for Code Generation	1-12

Modeling in Simulink Coder

2

Configure a Model for Code Generation	2-2
Supported Products and Block Usage	2-4
Related Products	2-4
Simulink Built-In Blocks That Support Code	
Generation	2-6
Simulink Block Data Type Support Table	2-27
Block Set Support for Code Generation	2-27
Modeling Semantic Considerations	2-28
Data Propagation	2-28
Sample Time Propagation	2-30
Latches for Subsystem Blocks	2-31
Block Execution Order	2-31

Algebraic Loops	2-33
Modeling Guidelines for Blocks	2-36
Modeling Guidelines for Subsystems	2-37
Modeling Guidelines for Charts	2-39
Modeling Guidelines for MATLAB Functions	2-41
Modeling Guidelines for Model Configuration	2-42

Timers in Simulink Coder

3

Absolute and Elapsed Time Computation	3-2
About Timers	3-2
Timers for Periodic and Asynchronous Tasks	3-3
Allocation of Timers	3-3
Integer Timers in Generated Code	3-3
Elapsed Time Counters in Triggered Subsystems	3-4
Access Timers Programmatically	3-5
About Timer APIs	3-5
C API for S-Functions	3-5
TLC API for Code Generation	3-7
Generate Code for an Elapsed Time Counter	3-9
Absolute Time Limitations	3-12

Time-Based Scheduling in Simulink Coder

4

Time-Based Scheduling and Code Generation	4-2
Sample Time Considerations	4-2
Tasking Modes	4-2

Model Execution and Rate Transitions	4-4
Execution During Simulink Model Simulation	4-5
Model Execution in Real Time	4-5
Single-Tasking Versus Multitasking Operation	4-6
Modeling for Single-Tasking Execution	4-8
Single-Tasking Mode	4-8
Build a Program for Single-Tasking Execution	4-8
Single-Tasking Execution	4-8
Modeling for Multitasking Execution	4-12
Multitasking and Pseudomultitasking Modes	4-12
Build a Program for Multitasking Execution	4-14
Execute Multitasking Models	4-14
Multitasking Execution	4-16
Handle Rate Transitions	4-20
Rate Transitions	4-20
Data Transfer Problems	4-21
Data Transfer Assumptions	4-22
Rate Transition Block Options	4-22
Automatic Rate Transition	4-25
Visualize Inserted Rate Transition Blocks	4-26
Periodic Sample Rate Transitions	4-28
Configure Time-Based Scheduling	4-34
Configure Start and Stop Times	4-34
Configure the Solver Type	4-34
Configure the Tasking Mode	4-35
Time-Based Scheduling Example Models	4-36
Optimize Memory Usage for Time Counters	4-36
Single-Rate Modeling (Bare Board, No OS)	4-40
Multirate Modeling in Single-Tasking Mode (Bare Board, no OS)	4-42
Multirate Modeling in Multitasking Mode (Bare Board, no OS)	4-44
Trade Determinism and Data Integrity to Improve System Performance	4-46

Asynchronous Events	5-2
Asynchronous Support	5-2
Block Library for Calls to an Example Real-Time Operating System	5-2
Access the Block Library for RTOS Integration	5-3
Generate Code Using Library Blocks for RTOS Integration	5-3
Examples and Additional Information	5-4
Generate Interrupt Service Routines	5-6
Connecting the Async Interrupt Block	5-6
Requirements and Restrictions	5-7
Performance Considerations	5-7
Using the Async Interrupt Block in Simulation and Code Generation	5-7
Dual-Model Approach: Simulation	5-8
Dual-Model Approach: Code Generation	5-9
Spawn and Synchronize Execution of RTOS Task	5-14
Pass Asynchronous Events in RTOS as Input To a Referenced Model	5-30
Rate Transitions and Asynchronous Blocks	5-37
About Rate Transitions and Asynchronous Blocks	5-37
Handle Rate Transitions for Asynchronous Tasks	5-39
Handle Multiple Asynchronous Interrupts	5-39
Timers in Asynchronous Tasks	5-42
Create a Customized Asynchronous Library	5-45
About Implementing Asynchronous Blocks	5-45
Async Interrupt Block Implementation	5-46
Task Sync Block Implementation	5-50
asynclib.tlc Support Library	5-51
Import Asynchronous Event Data for Simulation	5-53
Capabilities	5-53
Input Data Format	5-53

Example	5-53
Asynchronous Support Limitations	5-57
Asynchronous Task Priority	5-57
Convert an Asynchronous Subsystem into a Model	
Reference	5-57

Subsystems in Simulink Coder

6

Code Generation of Subsystems	6-2
Subsystem Code Dependence	6-3
Generate Code and Executables for Individual	
Subsystem	6-4
Subsystem Build Limitations	6-6
Inline Subsystem Code	6-7
Configure Subsystem to Inline Code	6-7
Exceptions to Inlining	6-8
Generate Subsystem Code as Separate Function and	
Files	6-10
Generate Reusable Function for Identical Subsystems	
Within a Model	6-11
Considerations for Function Packaging Options Auto and	
Reusable function	6-13
Code Reuse for Subsystems with Mask Parameters ...	6-13
Optimize Code for Identical Nested Subsystems	6-14
Generate Reusable Code for Subsystems Containing S-	
Function Blocks	6-15
Generate Reusable Code from Stateflow Charts	6-16
Code Reuse Limitations for Subsystems	6-17
Blocks That Prevent Code Reuse	6-18

Code Reuse Limitations for Subsystems Shared Across Referenced Models	6-18
Code Reuse For Subsystems Shared Across Models	6-20
Reusable Library Subsystem	6-21
Code Generation of a Reusable Library Subsystem	6-21
Reusable Library Subsystem Code Placement and Naming	6-22
Reusable Library Subsystem in the Top Model	6-22
Reusable Library Subsystem Connected to Root Output	6-22
Code Generation of Constant Parameters	6-23
Shared Constant Parameters for Code Reuse	6-24
Suppress Shared Constants in the Generated Code	6-25
Shared Constant Parameters Limitations	6-27
Generate Reusable Code for Subsystems Shared Across Models	6-28
Create a reusable library subsystem.	6-28
Create the example model.	6-31
Set configuration parameters of the top model.	6-33
Create and propagate a configuration reference.	6-33
Generate and view the code.	6-34
Determine Why Subsystem Code Is Not Reused	6-36
Review Subsystems Section of HTML Code Generation Report	6-36
Compare Subsystem Checksum Data	6-36

Code Generation of Functions and Function Callers in Simulink Coder

7

Modeling Functions and Callers for Code Generation	7-2
Functions and Callers	7-2
Input and Output Arguments	7-2

Function and Function Caller Definitions Across Models	7-3
Code Generation Files	7-3
Generate Code for Functions and Callers	7-6
Generate Code for the Function Definition	7-6
Generate Code for the Caller Definition	7-8

Referenced Models in Simulink Coder

8

Code Generation of Referenced Models	8-2
Generate Code for Referenced Models	8-4
About Generating Code for Referenced Models	8-4
Create and Configure the Subsystem	8-4
Convert Model to Use Model Referencing	8-7
Generate Model Reference Code for a GRT Target	8-11
Work with Code Generation Folders	8-13
Code Generation Folder Structure for Model Reference Targets	8-15
Configure Referenced Models	8-16
Build Model Reference Targets	8-17
Reduce Change Checking Time	8-17
Simulink Coder Model Referencing Requirements	8-18
Configuration Parameter Requirements	8-18
Naming Requirements	8-21
Custom Target Requirements	8-21
Storage Classes for Signals Used with Model Blocks	8-22
Storage Classes for Parameters Used with Model Blocks	8-22
Signal Name Mismatches Across Model Reference Boundary	8-23
Inherited Sample Time for Referenced Models	8-25

Customize Library File Suffix and File Type	8-27
Reusable Code and Referenced Models	8-28
General Considerations	8-28
Code Reuse and Model Blocks with Root Inport or Outport Blocks	8-28
Simulink Coder Model Referencing Limitations	8-32
Customization Limitations	8-32
Data Logging Limitations	8-32
State Initialization Limitation	8-33
Reusability Limitations	8-33
S-Function Limitations	8-34
Simulink Tool Limitations	8-34
Subsystem Limitations	8-34
Target Limitations	8-34
Other Limitations	8-35

Combined Models in Simulink Coder

9

Combined Models	9-2
Control Ownership of Data	9-3
Use GRT with Reusable Function Packaging to Combine Models	9-4
Share Data Across Models	9-4
Timing Issues	9-4
Data Logging and External Mode Support	9-5

Configure Model Parameters for Simulink Coder

10

Platform Options for Development and Deployment ..	10-2
Configure Target Hardware	10-3
Identify the Hardware Board	10-4

Identify the Device Vendor	10-5
Identify the Device Type	10-5
Register Additional Device Vendor and Device Type Values	10-5
Set Bit Lengths for Device Data Types	10-8
Set Byte Ordering Used By Device	10-10
Set Quotient Rounding Behavior for Signed Integer Division	10-10
Set Arithmetic Right Shift Behavior for Signed Integers	10-11
Update Release 14 Hardware Configuration	10-11
Configure Production Hardware Characteristics . . .	10-13
Configure Test Hardware Characteristics	10-14
Control the Location for Generated Files	10-15
Control Generated Files Location Used for Simulation	10-17
Control the Location for Code Generation Files	10-19
Override Build Folder Settings for Current Session . . .	10-21

Model Protection in Simulink Coder

11

Protect a Referenced Model	11-2
Requirements for Protecting a Model	11-3
Harness Model	11-4
Protected Model Report	11-5
Code Generation Support in a Protected Model	11-6
Protected Model Requirements to Support Code Generation	11-6
Protected Model File	11-8

Create a Protected Model	11-10
Protected Model Creation Settings	11-15
Open Read-Only View of Model	11-16
Simulate	11-16
Use Generated Code	11-16
Create a Protected Model with Multiple Targets	11-18
Use a Protected Model with Multiple Targets	11-19
Test the Protected Model	11-20
Save Base Workspace Definitions	11-22
Package a Protected Model	11-23
Specify Custom Obfuscator for Protected Model	11-24
Define Callbacks for Protected Model	11-26
Creating Callbacks	11-26
Defining Callback Code	11-27
Create a Protected Model with Callbacks	11-27

Component Initialization, Reset, and Termination in Simulink Coder

12

Generate Code That Responds to Initialize, Reset, and Terminate Events	12-2
Generate Code for Initialize and Terminate Events ...	12-2
Generate Code for Reset Events	12-7
Event Names and Code Aggregation	12-9
Limitations	12-12

Stateflow Blocks in Simulink Coder

13

Code Generation of Stateflow Blocks	13-2
Comparison of Code Generation Methods	13-2
Generate Reusable Code for Atomic Subcharts	13-6
How to Generate Reusable Code for Linked Atomic Subcharts	13-6
How to Generate Reusable Code for Unlinked Atomic Subcharts	13-7
Generate Reusable Code for Unit Testing	13-8
Goal of the Tutorial	13-8
Convert a State to an Atomic Subchart	13-9
Specify Code Generation Parameters	13-10
Generate Code for Only the Atomic Subchart	13-11
Inline State Functions in Generated Code	13-14
Inlined Generated Code for State Functions	13-14
How to Set the State Function Inline Option	13-16
Best Practices for Controlling State Function Inlining	13-16
Air-Fuel Ratio Control System with Stateflow Charts	13-17

Data, Function, and File Definition

Data Representation in Simulink Coder

14

Access Signal, State, and Parameter Data During Execution	14-3
Default Data Structures in the Generated Code	14-16
Use the Real-Time Model Data Structure	14-19

Use Enumerated Data in Generated Code	14-22
Enumerated Data Types	14-22
Specify Integer Data Type for Enumeration	14-22
Customize Enumerated Data Type	14-24
Control Enumerated Type Implementation in Generated Code	14-28
Type Casting for Enumerations	14-29
Enumerated Type Limitations	14-30
Data Stores in Generated Code	14-31
About Data Stores	14-31
Generate Code for Data Store Memory Blocks	14-31
Storage Classes for Data Store Memory Blocks	14-32
Data Store Buffering in Generated Code	14-34
Structures in Generated Code Using Data Stores ...	14-38
Explore Example Model	14-38
Configure Data Store	14-38
Write to Data Store Elements	14-39
Generate Code with Data Store Structure	14-41
Specify Single Precision	14-42
Block Parameter Representation in the Generated Code	14-43
Default Parameter Representation	14-43
Override Default Parameter Behavior by Creating Global Variables in the Generated Code	14-45
Parameter Object Configuration Quick Reference Diagram	14-47
Preservation of Expressions	14-47
Loss of Parameter Tunability	14-48
Configure Block Parameter Tunability for Rapid Prototyping	14-52
Tune Phase Parameter of Sine Wave Block During Code Execution	14-54
Create Tunable Calibration Parameter in the Generated Code	14-56
Represent Block Parameter as Tunable Global Variable	14-56

Configure Accessibility of Signal Data	14-58
Programmatic Interfaces for Tuning Parameters	14-59
Set Tunable Parameter Minimum and Maximum Values	14-59
Considerations for Other Modeling Goals	14-59
Specify Instance-Specific Parameter Values for Reusable Referenced Model	14-61
Pass Parameter Data to Referenced Model Entry-Point Functions as Arguments	14-61
Control Data Types of Model Arguments and Argument Values	14-73
Parameter Data Types in the Generated Code	14-76
Significance of Parameter Data Types	14-76
Parameter Data Type Mismatch	14-77
Considerations for Other Modeling Patterns	14-78
Generate Efficient Code by Specifying Data Types for Block Parameters	14-81
Eliminate Unnecessary Typecasts and Shifts by Matching Data Types	14-81
Reduce Memory Consumption by Storing Parameter Value in Small Data Type	14-84
Reuse Parameter Data in Different Data Type Contexts	14-90
Organize Block Parameter Values into Structures in the Generated Code	14-94
Creating Tunable Parameter Structures	14-94
Structures of Parameters	14-95
Structure Padding	14-98
Switch Between Sets of Parameter Values During Simulation and Code Execution	14-100
Signal Representation in Generated Code	14-109
Signal Storage Concepts	14-110
Signals with Auto Storage Class	14-112
Signals with Test Points	14-114
Symbolic Naming Conventions for Signals	14-115
Summary of Signal Storage Class Options	14-116

Interfaces for Monitoring Signals	14-117
Share Data Between Code Generated from Simulink, Stateflow, and MATLAB	14-117
Control Signals and States in Code by Applying Storage	
Classes	14-120
Storage Classes for Signals and States	14-121
Use Model Data Editor to Configure Data Interface .	14-124
Signal Objects for Code Generation	14-125
Create and Configure Signal Object for Code Generation	14-125
Programmatically Create and Configure Signal Object for Code Generation	14-126
Apply Storage Classes Directly to Signal Lines, Block States, and Outport Blocks	14-126
Programmatically Apply Storage Classes Directly to Signals, States, and Outport Blocks	14-127
Resolve Conflicts in Configuration of Signal Objects	14-128
 Design Data Interface by Configuring Inport and Outport Blocks	14-131
 Group Signals into Structures in the Generated Code	
Using Buses	14-136
Import or Export Structure Variable and Definition .	14-136
Model Data Specifications with Bus Signals	14-138
Arrays of Structures	14-139
Structure Padding	14-139
 Generate Efficient Code for Bus Signals	14-140
Code Efficiency for Bus Signals	14-140
Set Bus Diagnostics	14-141
Optimize Virtual and Nonvirtual Buses	14-141
 Maximize Signal Storage Optimization	14-144
 Control Signal and State Initialization in the Generated Code	14-145
Signal and State Initialization in the Generated Code	14-145
Generate Tunable Initial Conditions	14-147
Generate Tunable Initial Condition Structure for Bus Signal	14-150

Continuous Block State Naming in Generated Code	14-156
Default Block State Naming Convention	14-156
Define User Block State Names	14-157
Discrete Block State Naming in Generated Code . . .	14-158
Default Block State Naming Convention	14-159
Define User Block State Names	14-160
Signal Processing with Fixed-Point Data	14-163
Optimize Generated Code Using Fixed-Point Data with Simulink®, Stateflow®, and MATLAB®	14-165
Declare Workspace Variables as Tunable Parameters Using the Model Parameter Configuration Dialog Box	14-166
Declare Existing Workspace Variables as Tunable Parameters	14-166
Declare New Tunable Parameters	14-167
Set Tunable Parameter Code Generation Options . .	14-167
Programmatically Declare Workspace Variables as Tunable Parameters	14-168

Entry-Point Functions and Scheduling in Simulink Coder

15

Entry-Point Functions and Scheduling	15-2
Generate Reentrant Code from Top-Level Models	15-4
Generate C++ Class Interface to Model or Subsystem Code	15-6
Generate C++ Class Interface to Model Code	15-6
Generate C++ Class Interface to Nonvirtual Subsystem Code	15-7
C++ Class Interface Limitations	15-8
Execution of Code Generated from a Model	15-9
Program Execution	15-10

Program Timing	15-10
External Mode Communication	15-11
Data Logging in Single-Tasking and Multitasking Model Execution	15-12
Non-Real-Time Single-Tasking Systems	15-13
Non-Real-Time Multitasking Systems	15-13
Real-Time Single-Tasking Systems	15-15
Real-Time Multitasking Systems	15-16
Multitasking Systems Using Real-Time Tasking Primitives	15-18
Rapid Prototyping and Embedded Model Execution Differences	15-19
Rapid Prototyping Model Functions	15-21

Code Generation

Configuration for Simulink Coder

16

Code Generation Configuration	16-2
Open the Model Configuration for Code Generation	16-2
Configure Code Generation Parameters for Model Programmatically	16-4
Check Model and Configuration for Code Generation	16-6
Check Model During Code Generation	16-7
Application Objectives Using Code Generation Advisor	16-8
High-Level Code Generation Objectives	16-9
Configure Model for Code Generation Objectives Using Code Generation Advisor	16-10
Configure Model for Code Generation Objectives Using Configuration Parameters Dialog Box	16-11

Simulink Coder Model Advisor Checks for Standards and Code Efficiency	16-12
Configure Code Comments	16-13
Construction of Generated Identifiers	16-14
Identifier Name Collisions and Mangling	16-15
Identifier Name Collisions with Referenced Models ..	16-15
Specify Identifier Length to Avoid Naming Collisions	16-16
Specify Reserved Names for Generated Identifiers ..	16-17
Reserved Keywords	16-18
C Reserved Keywords	16-18
C++ Reserved Keywords	16-19
Reserved Keywords for Code Generation	16-19
Code Generation Code Replacement Library Keywords	16-20
Debug	16-22

Target Configuration for Simulink Coder

17

Select a System Target File	17-2
Target Platforms to Consider	17-2
System Target Files and Code Styles	17-2
Available System Target Files	17-3
System Target Files That Support Nonzero Start Time	17-6
Select a System Target File	17-7
Select a System Target File Programmatically	17-8
Specify Target Interfaces	17-10
Interface Dependencies	17-11
Configure Standard Math Library for Target System	17-15

System Target Files and Code Generation Features .	17-18
Compare Generated Code Features by Target	17-19
Compare Generated Code Features by Product	17-22
Template Makefiles and Make Options	17-26
Types of Template Makefiles	17-26
Specify Template Makefile Options	17-27
Template Makefiles for UNIX Platforms	17-27
Template Makefiles for the Microsoft Visual C++ Compiler	17-28
Template Makefiles for the LCC Compiler	17-30
Custom Targets	17-32

18 Internationalization Support in Simulink Coder

Internationalization and Code Generation	18-2
Locale Settings	18-2
Prepare to Generate Code for Mixed Languages and Locales	18-2
XML Escape Sequence Replacements	18-3
Character Set Limitations	18-3

19 Source Code Generation in Simulink Coder

Initiate Code Generation	19-2
Model and Test Environment	19-3
About This Example	19-3
Functional Design of the Model	19-4
View the Top Model	19-4
View the Subsystems	19-5
Simulation Test Environment	19-6
Run Simulation Tests	19-11
Key Points	19-12

Learn More	19-13
Configure Model and Generate Code	19-14
About This Example	19-14
Configure the Model for Code Generation	19-15
Save Your Model Configuration as a MATLAB Function	19-16
Check Model Conditions and Configuration Settings .	19-17
Generate Code for the Model	19-17
Review the Generated Code	19-17
Generate an Executable	19-19
Key Points	19-19
Configure Data Interface	19-21
About This Example	19-21
Declare Data	19-21
Use Data Objects	19-22
Add New Data Objects	19-25
Enable Data Objects for Generated Code	19-26
Effects of Simulation on Data Typing	19-26
Manage Data	19-28
Key Points	19-29
Call External C Functions	19-30
About This Example	19-30
Include External C Functions in a Model	19-31
Create a Block That Calls a C Function	19-31
Validate External Code in the Simulink Environment	19-33
Validate C Code as Part of a Model	19-34
Call a C Function from Generated Code	19-36
Key Points	19-36
Reload Generated Code	19-37
Generated Source Files and File Dependencies	19-38
About Generated Files and File Dependencies	19-38
Dependencies When Interfacing Custom and Generated Code	19-39
Dependencies of the Generated Code	19-49
Specify Include Paths in Generated Source Files	19-54
Files and Folders Created During Build Process	19-57
Files Created During the Build Process	19-57

Folders Used by the Build Process	19-62
Enable Build Process When Folder Names Have Spaces	19-65
Troubleshooting Errors When Folder Names Have Spaces	19-68
How Code Is Generated from a Model	19-71
Model Compilation	19-71
Code Generation	19-71
Code Generation of Matrices and Arrays	19-73
Code Generator Matrix Parameters	19-74
Internal Data Storage for Complex Number Arrays ..	19-76
Generated Code Considerations	19-77
Requirements for Signed Integer Representation	19-77
Default Folder for Code Generation	19-77
About Shared Utility Code	19-78
Controlling Shared Utility Code Placement	19-79
Avoid Duplicate Header Files for Exported Data	19-79
rtwtypes.h and Shared Utility Code	19-81
Incremental Shared Utility Code Generation and Compilation	19-82
Shared Utility Checksum	19-83
About the Shared Utility Checksum	19-83
View the Shared Utility Checksum Hash Table	19-83
Relate the Shared Utility Checksum to Configuration Parameters	19-85
Shared Fixed-Point Utility Functions	19-88
Share User-Defined Data Types Across Models	19-90
About Sharing Data Types	19-90
Example: Sharing Simulink Data Type Objects	19-91
Example: Sharing Enumerated Data Types	19-92

Cross-Release Shared Code Reuse	19-95
Workflow for Shared Code Reuse	19-95
Shared Code Files That Require Editing	19-96
Cross-Release Code Integration	19-98
Workflow	19-98
Limitations	19-101
Incorporate Model Reference Code	19-102
Simulink.Bus Support	19-102
Parameter Tuning	19-104
Generate Code Using Simulink® Coder™	19-106

Report Generation in Simulink Coder

20

Reports for Code Generation	20-2
HTML Code Generation Report Location	20-2
HTML Code Generation Report for Referenced Models	20-3
HTML Code Generation Report Extensions	20-3
Generate a Code Generation Report	20-5
Generate Code Generation Report After Build Process	20-6
Open Code Generation Report	20-8
Limitation	20-8
Generate Code Generation Report Programmatically	20-10
View Code Generation Report in Model Explorer ...	20-11
Package and Share the Code Generation Report	20-13
Package the Code Generation Report	20-13
View the Code Generation Report	20-14
Document Generated Code with Simulink Report Generator	20-15
Generate Code for the Model	20-16

Open the Report Generator	20-16
Set Report Name, Location, and Format	20-18
Include Models and Subsystems in a Report	20-19
Customize the Report	20-20
Generate the Report	20-21

Code Replacement in Simulink Coder

21

What Is Code Replacement?	21-2
Code Replacement Libraries	21-3
Code Replacement Terminology	21-5
Code Replacement Limitations	21-7
Choose a Code Replacement Library	21-8
About Choosing a Code Replacement Library	21-8
Explore Available Code Replacement Libraries	21-8
Explore Code Replacement Library Contents	21-8
Replace Code Generated from Simulink Models	21-10

Deployment

Desktops in Simulink Coder

22

Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File	22-2
About Rapid Simulation	22-2
Rapid Simulation Advantage	22-2
General Rapid Simulation Workflow	22-3
Identify Rapid Simulation Requirements	22-4
Configure Inports to Provide Simulation Source Data	22-6
Configure and Build Model for Rapid Simulation	22-6
Set Up Rapid Simulation Input Data	22-8

Scripts for Batch and Monte Carlo Simulations	22-18
Run Rapid Simulations	22-18
Tune Parameters Interactively During Rapid Simulation	22-29
Rapid Simulation Target Limitations	22-32

Accelerate Simulation, Reuse Code, or Protect

Intellectual Property by Using S-Function Target .	22-33
About the S-Function Target	22-33
Create S-Function Blocks from a Subsystem	22-36
Tunable Parameters in Generated S-Functions	22-40
System Target File	22-42
Checksums and the S-Function Target	22-42
Generated S-Function Compatibility	22-43
S-Function Target Limitations	22-43

Real-Time Systems in Simulink Coder

23

Deploy Algorithm Model for Real-Time Rapid

Prototyping	23-2
About Real-Time Rapid Prototyping	23-2
Goals of Real-Time Rapid Prototyping	23-2
Refine Code With Real-Time Rapid Prototyping	23-3

Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation

About Hardware-In-the-Loop Simulation	23-5
Set Up and Run HIL Simulations	23-6

External Code Integration in Simulink Coder

24

About External Code Integration	24-2
Choose External Code Integration Approach	24-3
Reuse Algorithmic Components in Generated Code	24-5

Deploy Algorithm Code Within a Target Environment	24-13
Export Generated Algorithm Code for Embedded Applications	24-18
Export Algorithm Executables for System Simulation	24-20
Modify External Code for Language Compatibility	24-20
Automate S-Function Generation with S-Function Builder	24-22
Macro Parameters	24-25
Build S-Function With Legacy Code Tool	24-27
Legacy Code Tool and Code Generation	24-27
Generate Inlined S-Function Files for Code Generation	24-27
Apply Code Style Settings to Legacy Functions	24-28
Address Dependencies on Files in Different Locations	24-29
Deploy S-Functions for Simulation and Code Generation	24-30
Use C++ Object Methods and Objects with Legacy Code Tool	24-31
Integrate External C++ Object Methods	24-31
Integrate External C++ Objects	24-34
External Code Integration Examples	24-37
Integrate External C Functions That Pass Input Arguments By Value Versus Address	24-38
Integrate External C Functions That Pass the Output Argument As a Return Argument	24-42
Integrate External C Functions That Pass Input and Output Arguments as Signals with a Fixed-Point Data Type	24-45
Integrate External C Functions with Instance-Specific Persistent Memory	24-48
Integrate External C Functions That Use Structure Arguments	24-51
Integrate External C Functions That Pass Input and Output Arguments as Signals with Complex Data	24-54
Integrate External C Functions That Pass Arguments That Have Inherited Dimensions	24-57
Integrate External C Functions That Implement Start and Terminate Actions	24-61
Integrate External C Functions That Pass Arguments as Multi-Dimensional Signals	24-64

Integrate External C Functions with a Block Sample Time Specified, Inherited, and Parameterized	24-67
Insert External C and C++ Code Into Stateflow Charts for Code Generation	24-72
Integrate External C Code Into Generated Code Using Custom Code Blocks and Model Configuration Parameters	24-74
Integrate External C Code Into Generated Code Using Custom Code Blocks and Model Configuration Parameters	24-77
Insert External C and C++ Code Into Stateflow Charts for Code Generation	24-79
Configure Model for External Code Integration	24-83
Insert Code into Model Custom Code Blocks	24-86
Custom Code Library	24-86
Embed Custom Code Directly Into MdlStart Function	24-90
Custom Code in Subsystems	24-92
Preserve User Files in Build Folder	24-93
Insert External Code into Stateflow Charts	24-94
Integrate External Code for Library Charts	24-94
Integrate External Code for All Charts	24-96
S-Functions and Code Generation	24-99
Classes of Problems Solved by S-Functions	24-99
Types of S-Functions	24-100
Basic Files Required for Implementation	24-102
Guidelines for Writing S-Functions	24-103
Write S-Function and TLC Files By Hand	24-104
Write Noninlined S-Function and TLC Files	24-104
Write Wrapper S-Function and TLC Files	24-106
Write Fully Inlined S-Functions	24-115
Write Fully Inlined S-Functions with mdlRTW Routine	24-116
Guidelines for Writing Inlined S-Functions	24-135
S-Functions That Support Expression Folding	24-135
S-Functions That Specify Port Scope and Reusability	24-146
S-Functions That Specify Sample Time Inheritance Rules	24-150
S-Functions That Support Code Reuse	24-152

S-Functions for Multirate Multitasking Environments 24-153

Build Support for S-Functions	24-160
About Build Support for S-Functions	24-160
Implicit Build Support	24-160
Specify Additional Source Files for an S-Function ..	24-161
Use TLC Library Functions	24-162
Precompile S-Function Libraries	24-162

Program Building, Interaction, and Debugging in Simulink Coder

25

Change Programming Language	25-2
Select and Configure Compiler or IDE	25-3
Compilers and the Build Process	25-3
Language Standards Compliance	25-3
Programming Language Considerations	25-4
C++ Language Limitations	25-5
Code Generator Optimizes Code by Relying on Undefined Behavior of C Language for Signed Integer Overflows	25-5
Choose and Configure Compiler on Microsoft Windows	25-6
Choose and Configure Compiler on UNIX	25-7
Include S-Function Source Code	25-7
Troubleshoot Compiler Issues	25-9
Compiler Version Mismatch Errors	25-9
Executable Program Generated for Model Produces Different Results from Model Simulation Results ..	25-9
Build Process Generates Expected Source Code But Executable Program Produces Incorrect Results ..	25-10
Compile-Time Issues	25-10
LCC Compiler Does Not Support Ampersands in Source Folder Paths	25-12
Configure Build Process	25-13
Choose a Build Process	25-13
Toolchain Approach	25-13

Upgrade Model to Use Toolchain Approach	25-15
Template Makefile Approach	25-19
Specify TLC Options	25-22
Build a Generic Real-Time Program	25-23
Building a Program	25-23
Working Folder	25-23
Build Folder and Code Generation Folders	25-23
Set Simulation Parameters	25-24
Select a Target Configuration	25-25
Set Code Generation Parameters	25-27
Build and Run a Program	25-32
Contents of the Build Folder	25-33
Initiate the Build Process	25-35
Build and Run a Program	25-36
How Executable Programs Are Built From Models	25-39
Build Process Steps	25-39
Customized Makefile Generation	25-39
Rebuild a Model	25-41
Control Regeneration of Top Model Code	25-43
Force Regeneration of Top Model Code	25-44
Reduce Build Time for Referenced Models	25-45
Parallel Building For Large Model Reference Hierarchies	25-45
Parallel Building Configuration Requirements	25-46
Build Models In a Parallel Computing Environment	25-46
Locate Parallel Build Logs	25-48
Relocate Code to Another Development Environment	25-51
Code Relocation	25-51
Package Code Using the Graphical User Interface	25-51
Package Code Using the Command-Line Interface	25-52
Integrate Generated Code with Custom Code in External Environment	25-56
packNGo Function Limitations	25-62
Executable Program Generation	25-63

Profile Code Performance	25-66
About Profiling Code Performance	25-66
How to Profile Code Performance	25-66
Embed Profiling Commands In Generated Code	25-68
Run Profiling Hooks for Generated Code	25-69
Profiling Limitation	25-70

26 **Host/Target Communication in Simulink Coder**

Set Up and Use Host/Target Communication Channel	26-2
What You Can Do With a Host/Target Communication Channel	26-2
Set Up an External Mode Communication Channel	26-3
Configure and Use External Mode	26-14
External Mode Compatible Blocks and Subsystems ..	26-34
External Mode Communication	26-37
Choose Communication Protocol for Client and Server	26-40
Use External Mode Programmatically	26-49
Animate Stateflow Charts in External Mode	26-54
External Mode Limitations	26-55

27 **Logging in Simulink Coder**

Log Program Execution Results	27-2
Log Data for Analysis	27-2
Configure State, Time, and Output Logging	27-9
Log Data with Scope and To Workspace Blocks	27-11
Log Data with To File Blocks	27-11
Data Logging Differences Between Single- and Multitasking	27-12

Data Interchange Using the C API in Simulink Coder

28

Exchange Data Between Generated and External Code	
Using C API	28-2
Generated C API Files	28-2
Generate C API Files	28-3
Description of C API Files	28-5
Generate C API Data Definition File for Exchanging Data with a Target System	28-20
C API Limitations	28-22
Use C API to Access Model Signals and States	28-24
Use C API to Access Model Parameters	28-30

ASAP2 Data Measurement and Calibration in Simulink Coder

29

Export ASAP2 File for Data Measurement and Calibration	29-2
What You Should Know	29-2
Targets Supporting ASAP2	29-3
Define ASAP2 Information	29-3
Generate an ASAP2 File	29-9
Structure of the ASAP2 File	29-12
Create a Host-Based ASAM-ASAP2 Data Definition File for Data Measurement and Calibration	29-13

Direct Memory Access to Generated Code for Simulink Coder

30

Access Memory in Generated Code Using Global Data Map	30-2
---	------

Performance

Optimizations for Generated Code in Simulink Coder

31

Optimization Parameters for Code Generation	31-3
Basic Optimization Parameters	31-3
Optimization Parameter Dependencies	31-4
Increase Code Generation Speed	31-7
Build a Model in Increments	31-7
Build Large Model Reference Hierarchies in Parallel ..	31-7
Minimize Memory Requirements During Code Generation	31-8
Generate Only Code	31-9
No Creation of a Code Generation Report	31-9
Design Tips for Optimizing Generated Code for Stateflow Objects	31-10
Do Not Access Machine-Parented Data In a Graphical Function	31-10
Be Explicit About the Inline Option of a Graphical Function	31-10
Avoid Using Multiple Edge-Triggered Events in Stateflow Charts	31-10
Combine Input Signals of a Chart Into a Single Bus Object	31-10
Use Charts with Discrete Sample Times	31-11

Control Compiler Optimizations	31-12
Optimization Tools and Techniques	31-13
Control Memory Allocation for Time Counters	31-15
Execution Profiling for Generated Code	31-16
Optimize Generated Code by Combining Multiple for Constructs	31-19
Subnormal Number Performance	31-22
Simulation Time With and Without Subnormal Numbers	31-23
Flush Subnormal Numbers to Zero	31-24
Remove Code From Floating-Point to Integer Conversions That Wraps Out-of-Range Values	31-27
Example Model	31-27
Generate Code Without Optimization	31-28
Generate Code with Optimization	31-29
Remove Code That Maps NaN to Integer Zero	31-30
Example Model	31-30
Generate Code	31-31
Generate Code with Optimization	31-32
Disable Nonfinite Checks or Inlining for Math Functions	31-34
Minimize Computations and Storage for Intermediate Results at Block Outputs	31-40
Expression Folding	31-40
Example Model	31-40
Generate Code	31-41
Enable Optimization	31-41
Generate Code with Optimization	31-42
Inline Invariant Signals	31-43
Optimize Generated Code Using Inline Invariant Signals	31-43
Inline Numeric Values of Block Parameters	31-47

Configure Loop Unrolling Threshold	31-53
Use memcpy Function to Optimize Generated Code for Vector Assignments	31-56
Example Model	31-57
Generate Code	31-58
Generate Code with Optimization	31-58
Generate Target Optimizations Within Algorithm Code	31-60
Remove Code for Blocks That Have No Effect on Computational Results	31-62
Eliminate Dead Code Paths in Generated Code	31-65
Floating-Point Multiplication to Handle a Net Slope Correction	31-68
Use Conditional Input Branch Execution	31-71
Optimize Generated Code for Complex Signals	31-77
Speed Up Linear Algebra in Code Generated from a MATLAB Function Block	31-79
Specify LAPACK Library	31-79
Write LAPACK Callback Class	31-79
Generate LAPACK Calls by Specifying a LAPACK Callback Class	31-80
Locate LAPACK Library in Execution Environment .	31-81
Optimize Memory Usage for Time Counters	31-83
Minimize Memory Requirements During Code Generation	31-88
Optimize Generated Code Using Boolean Data for Logical Signals	31-89
Reduce Memory Usage for Boolean and State Configuration Variables	31-92
Customize Stack Space Allocation	31-93

Optimize Generated Code Using <code>memset</code> Function . . .	31-95
Vector Operation Optimization	31-99
Enable and Reuse Local Block Outputs in Generated Code	31-102
Example Model	31-102
Generate Code Without Optimization	31-103
Enable Local Block Outputs and Generate Code . . .	31-103
Reuse Local Block Outputs and Generate Code	31-104

Verification

Simulation and Code Comparison in Simulink Coder

32

Simulation and Code Comparison	32-2
Configure Signal Data for Logging	32-2
Log Simulation Data	32-3
Run Executable and Load Data	32-5
Visualize and Compare Results	32-6
Compare States for Simulation and Code Generation . .	32-8

Numerical Consistency between Model and Generated Code

33

Numerical Consistency of Model and Generated Code	
Simulation Results	33-2
Numerical Consistency	33-2
Numerical Consistency in Complex Systems	33-3
Reasons for Block-Level Numerical Differences	33-5

Supported Hardware

Embedded Targets

34

Simulink Coder Supported Hardware	34-2
---	------

Customization

Build Process Integration in Simulink Coder

35

Control Build Process Compiling and Linking	35-2
Cross-Compile Code Generated on Microsoft Windows	35-4
Control Library Location and Naming During Build .	35-7
Library Control Parameters	35-7
Specify the Location of Precompiled Libraries	35-9
Control the Location of Model Reference Libraries ...	35-10
Control the Suffix Applied to Library File Names ...	35-11
Recompile Precompiled Libraries	35-13
Customize Post-Code-Generation Build Processing .	35-14
Workflow for Setting Up Customizations	35-14
Build Information Object	35-15
Program a Post Code Generation Command	35-16
Define a Post Code Generation Command	35-17
Customize Build Process with PostCodeGenCommand and Relocate Generated Code to an External Environment	35-18
Suppress Makefile Generation	35-20

Configure Generated Code with TLC	35-22
About Configuring Generated Code with TLC	35-22
Assigning Target Language Compiler Variables	35-22
Set Target Language Compiler Options	35-23
Use rtwmakecfg.m API to Customize Generated	
Makefiles	35-24
About the rtwmakecfg Function	35-24
Create the rtwmakecfg Function	35-25
Modify the Template Makefile for rtwmakecfg	35-27
Customize Build Process with STF_make_rtw_hook	
File	35-29
The STF_make_rtw_hook File	35-29
Conventions for Using the STF_make_rtw_hook File ..	35-29
STF_make_rtw_hook.m Function Prototype and	
Arguments	35-30
Applications for STF_make_rtw_hook.m	35-33
Control Code Regeneration Using	
STF_make_rtw_hook.m	35-34
Use STF_make_rtw_hook.m for Your Build Procedure	35-35
Customize Build Process with sl_customization.m ..	35-36
The sl_customization.m File	35-36
Register Build Process Hook Functions Using	
sl_customization.m	35-38
Variables Available for sl_customization.m Hook	
Functions	35-38
Example Build Process Customization Using	
sl_customization.m	35-39
Replace STF_rtw_info_hook Supplied Target Data ..	35-41
Customize Build to Use Shared Utility Code	35-42
Modify Template Makefiles to Support Shared	
Utilities	35-42

Run-Time Data Interface Extensions in Simulink Coder

36

Customize Generated ASAP2 File	36-2
About ASAP2 File Customization	36-2
ASAP2 File Structure on the MATLAB Path	36-2
Customize the Contents of the ASAP2 File	36-3
ASAP2 Templates	36-4
Customize Computation Method Names	36-6
Suppress Computation Methods for FIX_AXIS	36-7
Create a Transport Layer for External Communication	36-8
About Creating a Transport Layer for External Communication	36-8
Design of External Mode	36-8
External Mode Communications Overview	36-11
External Mode Source Files	36-12
Implement a Custom Transport Layer	36-16

37

Custom Target Development in Simulink Coder

About Embedded Target Development	37-2
Custom Targets	37-2
Types of Targets	37-2
Recommended Features for Embedded Targets	37-4
Sample Custom Targets	37-9
Target Development Folders, Files, and Builds	37-11
Folder and File Naming Conventions	37-11
Components of a Custom Target	37-12
Key Folders Under Target Root (mytarget)	37-17
Key Files in Target Folder (mytarget/mytarget)	37-19
Additional Files for Externally Developed Targets ...	37-22
Target Development and the Build Process	37-23

Customize System Target Files	37-29
Control Code Generation With the System Target File	37-29
System Target File Naming and Location Conventions	37-30
System Target File Structure	37-30
Define and Display Custom Target Options	37-37
Tips and Techniques for Customizing Your STF	37-44
Create a Custom Target Configuration	37-48
Customize Template Makefiles	37-62
Template Makefiles and Tokens	37-62
Invoke the make Utility	37-68
Structure of the Template Makefile	37-69
Customize and Create Template Makefiles	37-73
Custom Target Optional Features	37-79
Support Toolchain Approach with Custom Target ..	37-81
Support Model Referencing	37-83
About Model Referencing with a Custom Target	37-83
Declaring Model Referencing Compliance	37-84
Providing Model Referencing Support in the TMF	37-85
Controlling Configuration Option Value Agreement ..	37-88
Supporting the Shared Utilities Folder	37-88
Verifying Worker Configuration for Parallel Builds of	
Model Reference Hierarchies (Optional)	37-92
Preventing Resource Conflicts (Optional)	37-94
Support Compiler Optimization Level Control	37-95
About Compiler Optimization Level Control and Custom	
Targets	37-95
Declaring Compiler Optimization Level Control	
Compliance	37-95
Providing Compiler Optimization Level Control Support in	
the Target Makefile	37-96
Support C Function Prototype Control	37-97
About C Function Prototype Control and Custom	
Targets	37-97
Declaring C Function Prototype Control Compliance .	37-97
Providing C Function Prototype Control Support in the	
Custom Static Main Program	37-98

Support C++ Class Interface Control	37-100
About C++ Class Interface Control and Custom	
Targets	37-100
Declaring C++ Class Interface Control Compliance .	37-100
Providing C++ Class Interface Control Support in the	
Custom Static Main Program	37-101
 Support Concurrent Execution of Multiple Tasks . .	 37-102
 Interface to Development Tools	 37-104
About Interfacing to Development Tools	37-104
Template Makefile Approach	37-105
Interface to an Integrated Development Environment	37-105
 Device Drivers	 37-116

Model Architecture and Design

Modeling Environment for Simulink Coder

Design Models for Rapid Prototyping Deployment

When using Simulink® Coder™ to generate code for rapid prototyping, it is important to design your Simulink models with code generation in mind from the very beginning of the design process. Think about relevant design factors and issues such as:

In this section...
“Application Algorithms and Run-Time Environments” on page 1-2
“Software Execution Framework for Generated Code” on page 1-3
“Map Embedded System Architecture to Simulink Modeling Environment” on page 1-5
“Model Templates for Code Generation” on page 1-12

Application Algorithms and Run-Time Environments

Use Simulink to design models that represent application algorithms and run-time environments from which you intend to generate deployable code. Depending on your application, you might deploy code to an execution environment that consists of a combination of:

Execution Environment Components	Choices
Hardware	<ul style="list-style-type: none"> • Development computer • Rapid-prototyping board • Microprocessor • Microcontroller • FPGA • ASIC
Cores	<ul style="list-style-type: none"> • Single • Multiple
Operating system	<ul style="list-style-type: none"> • General-purpose • Real-time • None (bare metal)
Scheduling	<ul style="list-style-type: none"> • Single-tasking

Execution Environment Components	Choices
	<ul style="list-style-type: none"> • Multitasking • Interrupt driven • Concurrency • Provided by operating system • Generated from model
Application algorithm code	<ul style="list-style-type: none"> • Generated from model • External code

As you design models to generate C or C++ code for rapid prototyping or production deployment, keep in mind the execution environment. Generate code that meets implementation requirements and avoids potential design rework. As the preceding table reflects, the execution environment for code that you generate can range from relatively simple to complex. For example, a simple case is code that you generate from a single, single-tasking model that runs on a single-core microprocessor. A complex case is code that generate from a model partitioned to run as a distributed system on a multicore microprocessor and an FPGA.

Software Execution Framework for Generated Code

Part of an application execution environment is the software execution framework that is responsible for scheduling and running the generated code. That software can preexist, as in the case of an operating system and its scheduler, or you can code the software manually. The level of complexity varies depending on which of the following modeling and code generation scenarios applies:

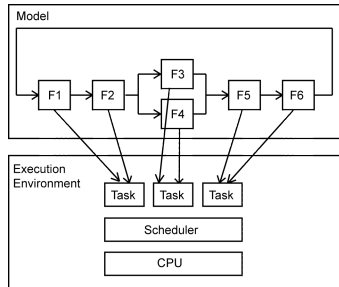
- Generate code from a single top model, which represents the algorithms intended to run in the execution environment.
- Generate code from a model, which represents part of an overall algorithm. You can mix the generated code with code written manually and code generated from other sources or releases of MathWorks® products.

Single Top Model

For a single top model, the software execution framework is responsible for running generated code the same way that Simulink simulates the model. Functions in the generated code are highly coordinated and optimized because Simulink is aware of

dependencies. The framework interfaces with code generated for the top model only. Code generated for a top model handles interfacing with code for referenced Model blocks.

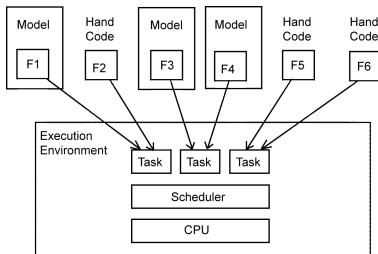
Consider the following example, where a single top model is mapped to tasks that run on a single-core CPU.



For this system, you map model clock rates to tasks that run on the hardware. You can choose for Simulink to map the rates implicitly or you can map them explicitly in your model. You can model latency effects resulting from how you map rates in a model to single-tasking or multitasking execution environments. Simulink schedules the tasks properly based on rates in the model and data dependencies between tasks. The code generator implements the same dependencies in the code that it generates. The software execution framework invokes generated entry-point functions at rates based on system timers and interrupts. The generated code executes in the same manner that Simulink simulates the model, and contains code dedicated to communicating data between functions running at different rates.

Multiple Top-Level Models

When you generate code from multiple top models separately and mix that code with code acquired in other ways, the execution environment of the application takes on more software execution framework responsibility. For this modeling scenario, you generate code for standalone, atomic reusable components.



With this scenario, Simulink is not aware of model dependencies. Functions in code generated from the different models are minimally coordinated and optimized. For example, the models might share generated utility functions. Potential optimizations that cross model boundaries are not possible. You must design the software execution framework taking into account dependencies between units of code, including execution order. For an application that requires concurrent execution across multiple cores, you must consider data latency effects across the cores.

The code generator helps you address software execution framework challenges, such as sharing global data and avoiding identifier conflicts. The code generated for a each model handles the interfacing for referenced Model blocks.

Map Embedded System Architecture to Simulink Modeling Environment

When designing models for rapid-prototyping deployment, think about these design factors.

“Modeling Algorithms” on page 1-6	Given initial state and input, a set of tasks or instructions that efficiently produce a correct result that you want.
“Modeling Interfaces” on page 1-6	Mechanisms that enable algorithm components to communicate and exchange information across component boundaries.
“Modeling Systems” on page 1-8	Collection of algorithm components that achieve a higher-level, domain-specific goal or result. Components often share resources.
“Modeling Run-Time Environments” on page 1-10	Framework that handles scheduling of system algorithm resources and execution.

Consider the following questions concerning modeling capabilities. Use the information listed with the questions as a guide as you design models. Designing a model with a specific run-time execution environment in mind can help you avoid rework and future conversion and maintenance costs.

Modeling Algorithms

Architecture Considerations	Modeling Considerations	Related Information
What is the system domain?	Product prerequisites (based on domains of components)	<ul style="list-style-type: none"> • “Supported Products and Block Usage” on page 2-4 (Simulink Coder) • “Simulink Control Design” (Simulink Control Design™) • “Model Signal Processing Systems” (Simulink) • “Signal Generation, Manipulation, and Analysis”(DSP System Toolbox™)
Does the system involve physical domains, such as mechanical, electrical, or hydraulic domains?	Physical systems	<ul style="list-style-type: none"> • “Model Physical Systems” (Simulink) • “Basic Principles of Modeling Physical Networks” (Simscape™) • “Essential Physical Modeling Techniques” (Simscape)
What aspects of your algorithm can you represent with blocks provided by MathWorks products? What blocks do you need to create?	Block usage, creation, and customization	<ul style="list-style-type: none"> • “Supported Products and Block Usage” on page 2-4 (Simulink Coder) • “External Code Integration” (Simulink Coder)
Does the architecture include state machine components?	Event-driven system	“Basic Approach for Modeling Event-Driven Systems” (Stateflow®)

Modeling Interfaces

Architecture Considerations	Modeling Considerations	Related Information
<ul style="list-style-type: none"> • What data must you represent in the generated code? • How do you need to represent input and output—data type, dimension, complexity? • How will the data change? 	Data representation	<ul style="list-style-type: none"> • “Interface Design” (Simulink) • “Data Representation”

Architecture Considerations	Modeling Considerations	Related Information
Where and how is data pulled into the system and pulled within the system?	Input	<ul style="list-style-type: none"> • “Comparison of Signal Loading Techniques” (Simulink)
<ul style="list-style-type: none"> • Where and how is data pushed within the system and out of the system? • What external triggers are necessary? 	Output	<ul style="list-style-type: none"> • “Inspect Signal Data with Simulation Data Inspector” (Simulink) • “Control Data Representation by Applying Custom Storage Classes”
<ul style="list-style-type: none"> • What functions do you need to define for each component? • What is the prototype for each entry-point function? 	Functions and function calls	<ul style="list-style-type: none"> • “Function and Class Interfaces”
Do you need to export functions that are invoked by controlling logic that is outside the model?	Function export	<ul style="list-style-type: none"> • “Export-Function Models” (Simulink) • “Export Generated Algorithm Code for Embedded Applications” on page 24-18
Does the system monitor signals or log data (for example, for calibration)?	C API and ASAP2 data exchange interfaces	<ul style="list-style-type: none"> • “Exchange Data Between Generated and External Code Using C API” on page 28-2 (Simulink Coder) • “Export ASAP2 File for Data Measurement and Calibration” on page 29-2 (Simulink Coder)
Do you need to replace code generated for functions or operators, for example, to optimize the code for specific hardware?	Code replacement	<ul style="list-style-type: none"> • “What Is Code Replacement?” on page 21-2 (Simulink Coder) • “What Is Code Replacement Customization?”
Is there a requirement for elaboration and future considerations?	Elaboration and future considerations	<ul style="list-style-type: none"> • “Interface Design” (Simulink)

Modeling Systems

Architecture Considerations	Modeling Considerations	Related Information
<ul style="list-style-type: none"> • What is the scope of the system? Controller? External environment or plant? Test harness? • How is the system partitioned into algorithm components (chunks of logic)? • Which components can you represent in Simulink? • Can you design components for reuse? What is the motivation for reuse (for example, division of labor or plug-n-play)? 	Componentization	<ul style="list-style-type: none"> • “Interface Design” (Simulink) • “Componentization Guidelines” (Simulink) • “Design Partitioning” (Simulink) • “Custom Libraries and Linked Blocks” (Simulink) • “Custom MATLAB Algorithms” (Simulink) • “Code Generation of Subsystems” on page 6-2 • “Code Generation of Referenced Models” on page 8-2 • “Code Generation of Stateflow Blocks” on page 13-2 (Simulink Coder)
<ul style="list-style-type: none"> • Do aspects of the system require unit testing? • Is a team of people collaborating on the project? • Do you need to protect intellectual property? 	Model referencing	<ul style="list-style-type: none"> • “Overview of Model Referencing” (Simulink) • “Componentization Guidelines” (Simulink) • “Code Generation of Referenced Models” on page 8-2 • “Generate Reusable Code for Unit Testing” on page 13-8
Are you modeling a client-server architecture?	Simulink Function and Caller blocks	<ul style="list-style-type: none"> • “Diagnostics Using a Client-Server Architecture” (Simulink) • “Simulink Functions” (Simulink)
Is relevant legacy or custom code available?	External code integration	“About External Code Integration” on page 24-2
Can you apply a reference architecture or reference components?	Model and project templates	<ul style="list-style-type: none"> • “Create a Template from a Model” (Simulink)

Architecture Considerations	Modeling Considerations	Related Information
		<ul style="list-style-type: none">• “Create a New Project Using Templates” (Simulink)
Can you reuse functions?	Function reuse	<ul style="list-style-type: none">• “Code Reuse For Subsystems Shared Across Models” on page 6-20 (Simulink Coder)• “Reusable Library Subsystem” on page 6-21 (Simulink Coder)• “Generate Reentrant Code from Top-Level Models” on page 15-4 (Simulink Coder)• “Reusable Code and Referenced Models” on page 8-28 (Simulink Coder)• “Generate Reusable Code for Atomic Subcharts” on page 13-6 (Simulink Coder)

Architecture Considerations	Modeling Considerations	Related Information
<ul style="list-style-type: none"> • Do components need to share access to global data? • Within the system, do state changes occur? In each case, how does the result get communicated? • Are there identifier (naming) issues to consider? 	Shared data	<ul style="list-style-type: none"> • “Local and Global Data Stores” (Simulink) • “Default Data Structures in the Generated Code” on page 14-16 (Simulink Coder) • “Storage Classes for Signals Used with Model Blocks” on page 8-22 (Simulink Coder) • “Shared Constant Parameters for Code Reuse” on page 6-24 (Simulink Coder) • “Data Stores in Generated Code” on page 14-31 (Simulink Coder) • “Create Data Objects for Code Generation with Data Object Wizard” • “Place Global Data Declarations and Definitions in Separate Files” • “Customize Generated Identifier Naming Rules”

Modeling Run-Time Environments

Architecture Considerations	Modeling Considerations	Related Information
<ul style="list-style-type: none"> • What level of control over run-time interfacing does your application require? • How much of your system can you represent in a model? 	Runtime interfacing	<ul style="list-style-type: none"> • “Execution of Code Generated from a Model” on page 15-9 • See Modeling Interfaces.
Is the system partitioned into concurrent components to	Concurrency	“Multicore Processor Targets” (Simulink)

Architecture Considerations	Modeling Considerations	Related Information
maximize parallelism? Which components?		
<ul style="list-style-type: none"> • Are components driven by an external clock? • What clock rates do system components use? • Do components use a single rate or multiple rates? 	Clocks and clock rates	“Interface Design” (Simulink)
<ul style="list-style-type: none"> • Are components in the system driven by clocks? • What clock rates do system components use? • Do components use a single rate or multiple rates? • What are the priorities of system tasks and functions? 	Time-based scheduling	<ul style="list-style-type: none"> • “Absolute and Elapsed Time Computation” on page 3-2 • “Time-Based Scheduling”
<ul style="list-style-type: none"> • Are components in the system driven by events (interrupts)? • What are the priorities of system tasks and functions? 	Event-based scheduling	<ul style="list-style-type: none"> • “Absolute and Elapsed Time Computation” on page 3-2 • “Event-Based Scheduling” • “Basic Approach for Modeling Event-Driven Systems” (Stateflow)
Does the system need to handle initialization, reset, or terminate events?	Initialization, reset, termination	<ul style="list-style-type: none"> • “Create a Model to Initialize and Terminate State” (Simulink) • “Generate Code That Responds to Initialize, Reset, and Terminate Events” on page 12-2

Architecture Considerations	Modeling Considerations	Related Information
<ul style="list-style-type: none"> • Is the system a single-tasking or multitasking system? • Are components required to execute in real time? • What are the execution order dependencies (sequencing) between components? • What are the time constraints for task and function execution? 	Task execution	<ul style="list-style-type: none"> • “Execution of Code Generated from a Model” on page 15-9 • “Modeling for Single-Tasking Execution” on page 4-8 • “Modeling for Multitasking Execution” on page 4-12
<ul style="list-style-type: none"> • If you know the processing platform, what is it? • Will the system run on a single-core or multicore processor? • Is the system a distributed system? • Is the processing platform hybrid or heterogeneous? • Does the architecture employ symmetric or asymmetric multiprocessing? If asymmetric, how is the platform software partitioned across CPUs? 	Processing platforms	“Multicore Processor Targets” (Simulink)

Model Templates for Code Generation

The code generator provides a set of built-in templates to use as a starting point to create models for common application designs. Use the templates to create models that are preconfigured to generate code for rapid-prototyping or embedded system applications.

Template	Description
Code Generation System	Basic model consisting of an Inport block and Output block.

Template	Description
Exported functions	Model for generating code from function-call subsystems. You can export each function-call subsystem separately by right-clicking a subsystem, selecting C/C++ Code > Export Functions , and clicking Build .
Fixed-step, multirate	Fixed-step model that uses multiple rates and consists of Inport blocks, an Outport block, and a Sum block. The model is configured to use a fixed-step discrete solver and to use two rates with Periodic sample time constraint set to Unconstrained and the Treat each discrete rate as a separate task option selected. Simulink inserts a Rate Transition block to handle the two sample rates.
Fixed-step, single rate	Fixed-step model that uses a single rate and consists of Inport blocks, an Outport block, and a Sum block. The model is configured to use a fixed-step discrete solver.

To create a model from a template:

- 1 On the MATLAB® home tab, click **Simulink**.
- 2 In the Simulink start page, expand **Embedded Coder**.
- 3 Select a template.
- 4 Click **Create**. A new model that uses the template contents and settings appears in the Simulink Editor window.

For more information, for example to create and use a template as a reference design, see “Create a Template from a Model”.

Related Examples

- “Supported Products and Block Usage” on page 2-4
- “Modeling Semantic Considerations” on page 2-28

Modeling in Simulink Coder

- “Configure a Model for Code Generation” on page 2-2
- “Supported Products and Block Usage” on page 2-4
- “Modeling Semantic Considerations” on page 2-28
- “Modeling Guidelines for Blocks” on page 2-36
- “Modeling Guidelines for Subsystems” on page 2-37
- “Modeling Guidelines for Charts” on page 2-39
- “Modeling Guidelines for MATLAB Functions” on page 2-41
- “Modeling Guidelines for Model Configuration” on page 2-42

Configure a Model for Code Generation

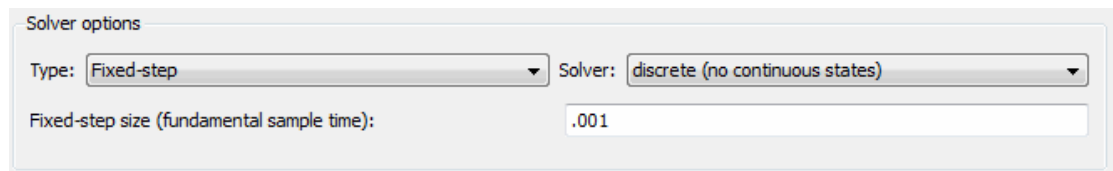
Model configuration parameters determine the method for generating the code and the resulting format.

- 1 Open `rtwdemo_throttlecntrl` and save a copy as `throttlecntrl` in a writable location on your MATLAB path.

Note: This model uses Stateflow software.

- 2 Open the Configuration Parameters dialog box **Solver** pane. To generate code for a model, you must configure the model to use a fixed-step solver. For this example, set the parameters as noted in the following table.

Parameter	Setting	Effect on Generated Code
Type	Fixed-step	Maintains a constant (fixed) step size, which is required for code generation
Solver	discrete (no continuous states)	Applies a fixed-step integration technique for computing the state derivative of the model
Fixed-step size	.001	Sets the base rate; must be the lowest common multiple of all rates in the system



- 3 Open the **Code Generation** pane and make sure that **System target file** is set to `grt.tlc`.

Note: The GRT (Generic Real-Time Target) configuration requires a fixed-step solver. However, the `rsim.tlc` system target file supports variable step code generation.

The system target file (STF) defines a target, which is an environment for generating and building code for execution on a certain hardware or operating system platform. For example, one property of a target is code format. The grt configuration requires a fixed step solver and the `rsim.tlc` supports variable step code generation.

- 4 Open the **Code Generation > Custom Code** pane, and under **Include list of additional**, select **Include directories**. In the **Include directories** text field, enter:

```
"$matlabroot$\toolbox\rtw\rtwdemos\EmbeddedCoderOverview\"
```

This directory includes files that are required to build an executable for the model.

- 5 Apply your changes and close the dialog box.

Supported Products and Block Usage

In this section...
“Related Products” on page 2-4
“Simulink Built-In Blocks That Support Code Generation” on page 2-6
“Simulink Block Data Type Support Table” on page 2-27
“Block Set Support for Code Generation” on page 2-27

Related Products

The following table summarizes MathWorks products that extend and complement Simulink Coder software. For information about these and other MathWorks products, see www.mathworks.com.

Product	Extends Code Generation Capabilities for ...
Aerospace Blockset™	Aircraft, spacecraft, rocket, propulsion systems, and unmanned airborne vehicles
Communications System Toolbox™	Physical layer of communication systems
Audio System Toolbox™	Audio processing systems
Computer Vision System Toolbox™	Video processing, image processing, and computer vision systems
Control System Toolbox™	Linear control systems
DSP System Toolbox	Signal processing systems
Embedded Coder®	Embedded systems, on-target rapid prototyping boards, microprocessors in mass production, and real-time simulators
Fixed-Point Designer™	Fixed-point systems
Fuzzy Logic Toolbox™	System designs based on fuzzy logic
Model-Based Calibration Toolbox™	Developing processes for systematically identifying optimal balance of engine performance, emissions, and fuel economy, and reusing statistical models for control design, hardware-in-the-loop (HIL) testing, or powertrain simulation

Product	Extends Code Generation Capabilities for ...
Model Predictive Control Toolbox™	Controllers that optimize performance of multi-input and multi-output systems that are subject to input and output constraints
Neural Network Toolbox™	Neural networks
Phased Array System Toolbox™	Sensor array systems in radar, sonar, wireless communications, and medical imaging applications
Simulink Desktop Real-Time™	Rapid prototyping or hardware-in-the-loop (HIL) simulation of control system and signal processing algorithms
Simscape Driveline™	Driveline (drivetrain) systems
Simscape Electronics™	Electronic and electromechanical systems
Simscape Fluids™	Hydraulic power and control systems
Simscape Multibody™	Three-dimensional mechanical systems
Simscape Power Systems™	Systems that generate, transmit, distribute, and consume electrical power
Simscape	Systems spanning mechanical, electrical, hydraulic, and other physical domains as physical networks
Simulink 3D Animation™	Systems with 3D visualizations
Simulink Design Optimization™	Systems requiring maximum overall system performance
Simulink Real-Time™	Rapid control prototyping, hardware-in-the-loop (HIL) simulation, and other real-time testing applications
Simulink Report Generator™	Automatically generating project documentation in a standard format
Simulink Verification and Validation™	Applications requiring automated requirements tracing, model standards compliance checking, and test harness generation
Stateflow	State machines and flow charts

Product	Extends Code Generation Capabilities for ...
System Identification Toolbox™	Systems constructed from measured input-output data Support exceptions: <ul style="list-style-type: none"> • Nonlinear IDNLGREY Model, IDDATA Source, IDDATA Sink, and estimator blocks • Nonlinear ARX models that contain custom regressors • <code>neuralnet</code> nonlinearities • <code>customnet</code> nonlinearities
Vehicle Network Toolbox™	CAN blocks for Accelerator and Rapid Accelerator simulations and code deployment on Windows®

Simulink Built-In Blocks That Support Code Generation

The following tables summarize code generator support for Simulink blocks. There is a table for each block library. For more detail, including data types each block supports, in the MATLAB Command Window, type `showblockdatatypetable`, or consult the block reference pages. For some blocks, the generated code might rely on `memcpy` or `memset` (`string.h`).

- Additional Math and Discrete: Additional Discrete
- Additional Math and Discrete: Increment/Decrement
- Continuous
- Discontinuities
- Discrete
- Logic and Bit Operations
- Lookup Tables
- Math Operations
- Model Verification
- Model-Wide Utilities
- Ports & Subsystems

- Signal Attributes
- Signal Routing
- Sinks
- Sources
- User-Defined

Additional Math and Discrete: Additional Discrete

Block	Support Notes
Fixed-Point State-Space	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.
Transfer Fcn Direct Form II	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.
Transfer Fcn Direct Form II Time Varying	
Unit Delay Enabled (Obsolete)	
Unit Delay Enabled External IC (Obsolete)	
Unit Delay Enabled Resettable (Obsolete)	
Unit Delay Enabled Resettable External IC (Obsolete)	
Unit Delay External IC (Obsolete)	
Unit Delay Resettable (Obsolete)	
Unit Delay Resettable External IC (Obsolete)	
Unit Delay With Preview Enabled (Obsolete)	
Unit Delay With Preview Enabled Resettable (Obsolete)	
Unit Delay With Preview Enabled Resettable External RV (Obsolete)	
Unit Delay With Preview Resettable (Obsolete)	

Block	Support Notes
Unit Delay With Preview Resettable External RV (Obsolete)	

Additional Math and Discrete: Increment/Decrement

Block	Support Notes
Decrement Real World	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.
Decrement Stored Integer	
Decrement Time To Zero	Supports code generation.
Decrement To Zero	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.
Increment Real World	
Increment Stored Integer	

Continuous

Block	Support Notes
Derivative	Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.
Integrator	
Integrator Limited	
PID Controller	
PID Controller (2DOF)	
Second-Order Integrator	
Second-Order Integrator Limited	In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select Analysis > Control Design > Model Discretizer . One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.
State-Space	
Transfer Fcn	
Transport Delay	
Variable Time Delay	
Variable Transport Delay	

Block	Support Notes
Zero-Pole	

Discontinuities

Block	Support Notes
Backlash	Supports code generation.
Coulomb and Viscous Friction	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.
Dead Zone	Supports code generation.
Dead Zone Dynamic	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.
Hit Crossing	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Quantizer	Supports code generation.
Rate Limiter	Cannot use inside a triggered subsystem hierarchy.
Rate Limiter Dynamic	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.

Block	Support Notes
Relay	Support code generation.
Saturation	
Saturation Dynamic	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.
Wrap To Zero	

Discrete

Block	Support Notes
Delay	Supports code generation.
Difference	<ul style="list-style-type: none"> The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option. Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Discrete Derivative	<ul style="list-style-type: none"> Depends on absolute time when used inside a triggered subsystem hierarchy. Supports code generation.
Discrete Filter	Support code generation.
Discrete FIR Filter	
PID Controller	<ul style="list-style-type: none"> Depends on absolute time when used inside a triggered subsystem hierarchy.
PID Controller (2DOF)	

Block	Support Notes
	<ul style="list-style-type: none"> Support code generation.
Discrete State-Space	Support code generation.
Discrete Transfer Fcn	
Discrete Zero-Pole	
Discrete-Time Integrator	Depends on absolute time when used inside a triggered subsystem hierarchy.
Enabled Delay	Supports code generation.
First-Order Hold	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Memory	Support code generation.
Resetable Delay	
Tapped Delay	
Transfer Fcn First Order	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.
Transfer Fcn Lead or Lag	
Transfer Fcn Real Zero	
Unit Delay	Support code generation.
Variable Integer Delay	
Zero-Order Hold	

Logic and Bit Operations

Block	Support Notes
Bit Clear	Support code generation.

Block	Support Notes
Bit Set	
Bitwise Operator	
Combinatorial Logic	
Compare to Constant	
Compare to Zero	
Detect Change	
Detect Decrease	
Detect Fall Negative	
Detect Fall Nonpositive	
Detect Increase	
Detect Rise Nonnegative	
Detect Rise Positive	
Extract Bits	
Interval Test	
Interval Test Dynamic	
Logical Operator	
Relational Operator	
Shift Arithmetic	

Lookup Tables

Block	Support Notes
Cosine	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit check box.

Block	Support Notes
Direct Lookup Table (n-D)	Support code generation.
Interpolation Using Prelookup	
1-D Lookup Table	
2-D Lookup Table	
n-D Lookup Table	
Lookup Table Dynamic	
Prelookup	
Sine	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.

Math Operations

Block	Support Notes
Abs	Support code generation.
Add	
Algebraic Constraint	Ignored during code generation.
Assignment	Support code generation.
Bias	
Complex to Magnitude-Angle	
Complex to Real-Imag	
Divide	
Dot Product	
Find Nonzero Elements	
Gain	

Block	Support Notes
Magnitude-Angle to Complex	
Math Function (10 ^u)	
Math Function (conj)	
Math Function (exp)	
Math Function (hermitian)	
Math Function (hypot)	
Math Function (log)	
Math Function (log10)	
Math Function (magnitude ²)	
Math Function (mod)	
Math Function (pow)	
Math Function (reciprocal)	
Math Function (rem)	
Math Function (square)	
Math Function (transpose)	
Matrix Concatenate	
MinMax	
MinMax Running Resettable	
Permute Dimensions	
Polynomial	
Product	
Product of Elements	
Real-Imag to Complex	
Reciprocal Sqrt	
Reshape	

Block	Support Notes
Rounding Function	
Sign	
Signed Sqrt	
Sine Wave Function	<ul style="list-style-type: none"> Does not refer to absolute time when configured for sample-based operation. Depends on absolute time when in time-based operation. Depends on absolute time when used inside a triggered subsystem hierarchy.
Slider Gain	Support code generation.
Sqrt	
Squeeze	
Subtract	
Sum	
Sum of Elements	
Trigonometric Function	Functions <code>asinh</code> , <code>acosh</code> , and <code>atanh</code> are not supported by all compilers. If you use a compiler that does not support those functions, the software issues a warning for the block and the generated code fails to link.
Unary Minus	Support code generation.
Vector Concatenate	
Weighted Sample Time Math	

Model Verification

Block	Support Notes
Assertion	Supports code generation.
Check Discrete Gradient	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets

Block	Support Notes
	often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Check Dynamic Gap	Support code generation.
Check Dynamic Lower Bound	
Check Dynamic Range	
Check Dynamic Upper Bound	
Check Input Resolution	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Check Static Gap	
Check Static Lower Bound	
Check Static Range	
Check Static Upper Bound	

Model-Wide Utilities

Block	Support Notes
Block Support Table	Ignored during code generation.
DocBlock	Uses the template symbol you specify for the Embedded Coder Flag block parameter to add comments to generated code. Requires an Embedded Coder license. For more information, see “Use a Simulink DocBlock to Add a Comment”.
Model Info	Ignored during code generation.
Timed-Based Linearization	
Trigger-Based Linearization	

Ports & Subsystems

Block	Support Notes
Atomic Subsystem	Support code generation.
CodeReuse Subsystem	
Configurable Subsystem	
Enable	
Enabled Subsystem	
Enabled and Triggered Subsystem	
For Each Subsystem	
For Iterator Subsystem	
Function-Call Feedback Latch	
Function-Call Generator	
Function-Call Split	
Function-Call Subsystem	
If	
If Action Subsystem	
Inport (In1)	
Model	
Model Variants	
Output (Out1)	
Resettable Subsystem	
Subsystem	
Switch Case	
Switch Case Action Subsystem	
Trigger	
Triggered Subsystem	

Block	Support Notes
Unit System Configuration	
Variant Subsystem	
While Iterator Subsystem	

Signal Attributes

Block	Support Notes
Bus to Vector	Support code generation.
Data Type Conversion	
Data Type Conversion Inherited	
Data Type Duplicate	
Data Type Propagation	
Data Type Scaling Strip	
IC	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Probe	Supports code generation.
Rate Transition	<ul style="list-style-type: none"> • Supports code generation. • Cannot use inside a triggered subsystem hierarchy.
Signal Conversion	Support code generation.
Signal Specification	
Unit Conversion	

Block	Support Notes
Weighted Sample Time	
Width	

Signal Routing

Block	Support Notes
Bus Assignment	Support code generation.
Bus Creator	
Bus Selector	
Data Store Memory	
Data Store Read	
Data Store Write	
Demux	
Environment Controller	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
From	Support code generation.
Goto	
Goto Tag Visibility	
Index Vector	
Manual Switch	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being

Block	Support Notes
	suitable for production code. Thus, blocks suitable for production code remain suitable.
Merge	When multiple signals connected to a Merge block have a non- Auto storage class, all non- Auto signals connected to that block must <i>be identically labeled and have the same storage class</i> . When Merge blocks connect directly to one another, these rules apply to all signals connected to Merge blocks in the group.
Multiport Switch	Support code generation.
Mux	
Selector	
Switch	
Variant Sink	
Variant Source	
Vector Concatenate	

Sinks

Block	Support Notes
Display	Ignored for code generation.
Floating Scope	
Outport (Out1)	Supports code generation.
Scope	Ignored for code generation.
Stop Simulation	<ul style="list-style-type: none"> Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable. Generated code stops executing when the stop condition is true.
Terminator	Supports code generation.

Block	Support Notes
To File	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
To Workspace	Ignored for code generation.
XY Graph	

Sources

Block	Support Notes
Band-Limited White Noise	Cannot use inside a triggered subsystem hierarchy.
Chirp Signal	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Clock	
Constant	Supports code generation.
Counter Free-Running	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

Block	Support Notes
Counter Limited	<ul style="list-style-type: none"> • The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option. • Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Digital Clock	<p>Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.</p>
Enumerated Constant	<p>Supports code generation.</p>
From File	<p>Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.</p>
From Spreadsheet	
From Workspace	<p>Ignored for code generation.</p>
Ground	<p>Support code generation.</p>

Block	Support Notes
Inport (In1)	
Pulse Generator	Cannot use inside a triggered subsystem hierarchy. Does not refer to absolute time when configured for sample-based operation. Depends on absolute time when in time-based operation.
Ramp	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Random Number	Supports code generation.
Repeating Sequence	<ul style="list-style-type: none"> • Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable. • Consider using the Repeating Sequence Stair or Repeating Sequence Interpolated block instead.
Repeating Sequence Interpolated	<ul style="list-style-type: none"> • The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option. • Cannot use inside a triggered subsystem hierarchy.
Repeating Sequence Stair	The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In

Block	Support Notes
	certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.
Signal Builder	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Signal Generator	
Sine Wave	<ul style="list-style-type: none"> • Depends on absolute time when used inside a triggered subsystem hierarchy. • Does not refer to absolute time when configured for sample-based operation. Depends on absolute time when in time-based operation.
Step	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Uniform Random Number	Supports code generation.
Waveform Generator	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

User-Defined

Block	Support Notes
Fcn	Support code generation.
Function Caller	
Interpreted MATLAB Function	Consider using the MATLAB Function block instead.
Level-2 MATLAB S-Function	Ignored during code generation.
MATLAB Function	Support code generation.
MATLAB System	
S-Function	S-functions that call into MATLAB are not supported for code generation.
S-Function Builder	
Simulink Function	Supports code generation.

Simulink Block Data Type Support Table

The Simulink Block Data Type Support table summarizes characteristics of blocks in the Simulink and Fixed-Point Designer block libraries, including whether or not they are recommended for use in production code generation. To view this table, in the MATLAB Command Window, type `showblockdatatypetable`, or consult the block reference pages.

Block Set Support for Code Generation

Several products that include blocks are available for you to consider for code generation. However, before using the blocks for one of these products, consult the documentation for that product to confirm which blocks support code generation.

Modeling Semantic Considerations

In this section...
“Data Propagation” on page 2-28
“Sample Time Propagation” on page 2-30
“Latches for Subsystem Blocks” on page 2-31
“Block Execution Order” on page 2-31
“Algebraic Loops” on page 2-33

Data Propagation

The first stage of code generation is compilation of the block diagram. This stage is analogous to that of a C or C++ program. The compiler carries out type checking and preprocessing. Similarly, the Simulink engine verifies that input/output data types of block ports are consistent, line widths between blocks are of expected thickness, and the sample times of connecting blocks are consistent.

The Simulink engine propagates data from one block to the next along signal lines. The data propagated consists of

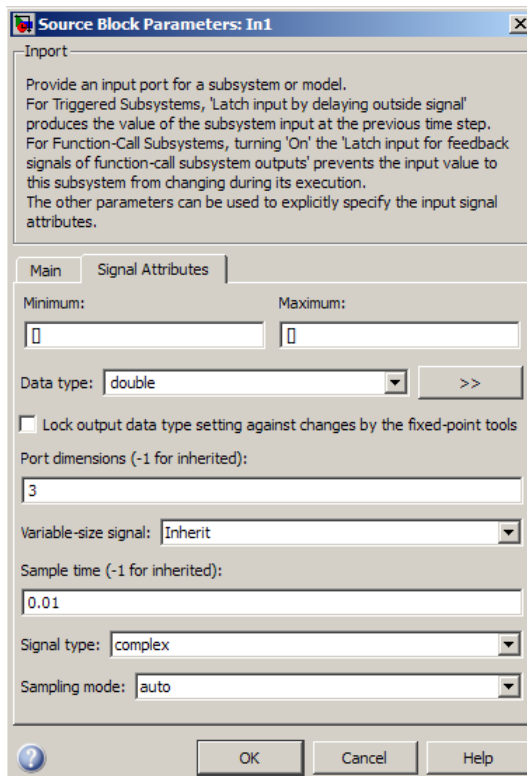
- Data type
- Line widths
- Sample times

You can verify what data types a Simulink block supports by typing

```
showblockdatatypetable
```

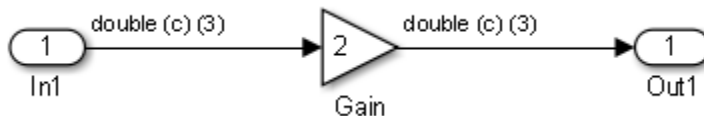
at the MATLAB prompt, or (from the Help browser) clicking the command above.

The Simulink engine typically derives signal attributes from a source block. For example, the Inport block's parameters dialog box specifies the signal attributes for the block.



In this example, the Inport block has a port width of 3, a sample time of .01 seconds, the data type is double, and the signal is complex.

This figure shows the propagation of the signal attributes associated with the Inport block through a simple block diagram.



In this example, the Gain and Outport blocks inherit the attributes specified for the Inport block.

Sample Time Propagation

Inherited sample times in source blocks (for example, a root inport) can sometimes lead to unexpected and unintended sample time assignments. Since a block may specify an inherited sample time, information available at the outset is often insufficient to compile a block diagram completely.

In such cases, the Simulink engine propagates the known or assigned sample times to those blocks that have inherited sample times but that have not yet been assigned a sample time. Thus, the engine continues to fill in the blanks (the unknown sample times) until sample times have been assigned to as many blocks as possible. Blocks that still do not have a sample time are assigned a default sample time.

For a completely deterministic model (one where no sample times are set using the above rules), you should explicitly specify the sample times of your source blocks. Source blocks include root inport blocks and blocks without input ports. You do not have to set subsystem input port sample times. You might want to do so, however, when creating modular systems.

An unconnected input implicitly connects to ground. For ground blocks and ground connections, the sample time is always constant (`inf`).

All blocks have an inherited sample time ($T_s = -1$). They are assigned a sample time of $(T_f - T_i)/50$.

Blocks Whose Outputs Have Constant Values

When you display sample time colors, by default, Constant blocks appear magenta in color to indicate that the block outputs have constant values during simulation. Downstream blocks whose output values are also constant during simulation, such as Gain blocks, similarly appear magenta if they use an inherited sample time. The code generated for these blocks depends in part on the tunability of the block parameters.

If you set **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** to **Inlined**, the block parameters are not tunable in the generated code. Because the block outputs are constant, the code generator eliminates the block code due to constant folding. If the code generator cannot fold the code, or if you select settings to disable constant folding, the block code appears in the model initialization function. The generated code is more efficient because it does not compute the outputs of these blocks during execution.

However, if you configure a block or model so that the block parameters appear in the generated code as tunable variables, the code generator represents the blocks in a different way. Block parameters are tunable if, for example:

- You set **Default parameter behavior** to **Tunable**. By default, numeric block parameters appear as tunable fields of a global parameter structure.
- You use a tunable parameter, such as a `Simulink.Parameter` object that uses a storage class other than `Auto`, as the value of one or more numeric block parameters. These block parameters are tunable regardless of the setting that you choose for **Default parameter behavior**.

If a block parameter is tunable, the generated code must compute the block outputs during execution. Therefore, the block code appears in the model `step` function. If the model uses multiple discrete rates, the block code appears in the output function for the fastest downstream rate that uses the block outputs.

Latches for Subsystem Blocks

When an Inport block is the signal source for a triggered or function-call subsystem, you can use latch options to preserve input values while the subsystem executes. The Inport block latch options include:

For	Use
Triggered subsystems	Latch input by delaying outside signal
Function-call subsystems	Latch input for feedback signals of function-call subsystem outputs

When you use **Latch input for feedback signals of function-call subsystem outputs** for a function-call subsystem, the code generator

- Preserves latches in generated code regardless of optimizations that might be set
- Places the code for latches at the start of a subsystem's output/update function

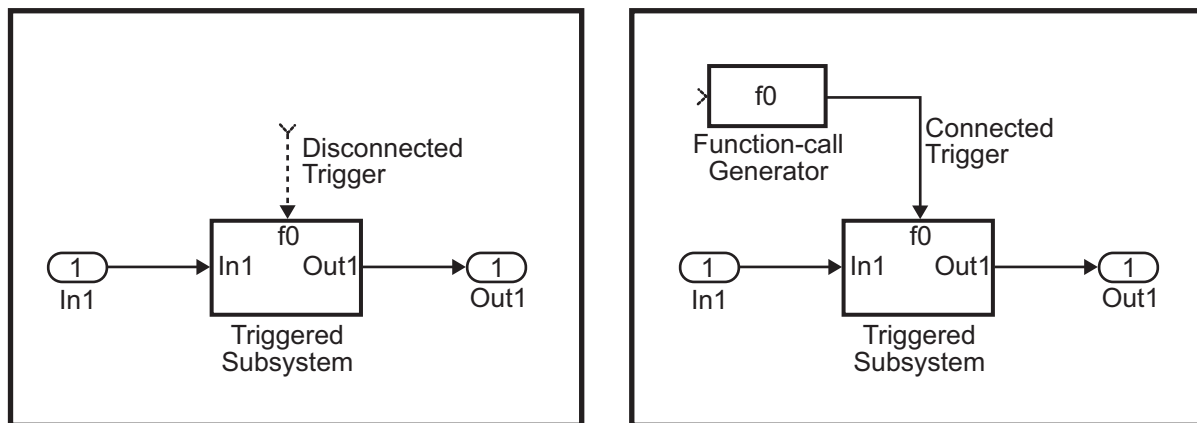
For more information on these options, see the description of the Inport block in the Simulink documentation.

Block Execution Order

Once the Simulink engine compiles the block diagram, it creates a `model.rtw` file (analogous to an object file generated from a C or C++ file). The `model.rtw` file contains

the connection information of the model, as well as the signal attributes. Thus, the timing engine in can determine when blocks with different rates should be executed.

You cannot override this execution order by directly calling a block (in handwritten code) in a model. For example, in the next figure the `disconnected_trigger` model on the left has its trigger port connected to ground, which can lead to the blocks inheriting a constant sample time. Calling the trigger function, `f()`, directly from user code does not work. Instead, you should use a function-call generator to specify the rate at which `f()` should be executed, as shown in the `connected_trigger` model on the right.



Instead of the function-call generator, you could use another block that can drive the trigger port. Then, you should call the model's main entry point to execute the trigger function.

For multirate models, a common use of the code generator is to generate code for individual models separately and then manually code the I/O between the generated code modules. This approach places the burden of data consistency between models on the developer of the models. Another approach is to let Simulink and the code generator maintain data consistency between rates and generate multirate code for use in a multitasking environment. The Rate Transition block is able to interface periodic and asynchronous signals. For a description of the Simulink Coder block libraries, see “Asynchronous Events” on page 5-2. For more information on multirate code generation, see “Modeling for Multitasking Execution” on page 4-12.

Algebraic Loops

Algebraic loops are circular dependencies between variables. This prevents the straightforward direct computation of their values. For example, in the case of a system of equations

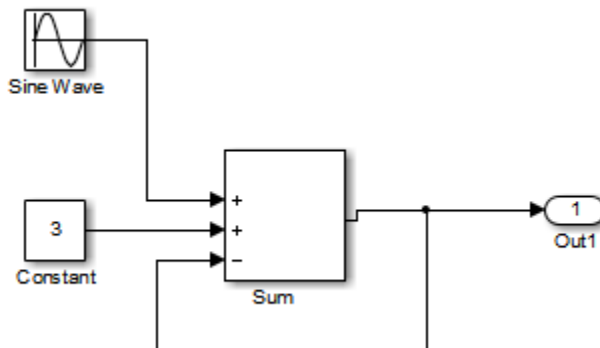
- $x = y + 2$
- $y = -x$

the values of x and y cannot be directly computed.

To solve this, either repeatedly try potential solutions for x and y (in an intelligent manner, for example, using gradient based search) or “solve” the system of equations. In the previous example, solving the system into an explicit form leads to

- $2x = 2$
- $y = -x$
- $x = 1$
- $y = -1$

An algebraic loop exists whenever the output of a block having direct feedthrough (such as Gain, Sum, Product, and Transfer Fcn) is fed back as an input to the same block. The Simulink engine is often able to solve models that contain algebraic loops, such as the next diagram.

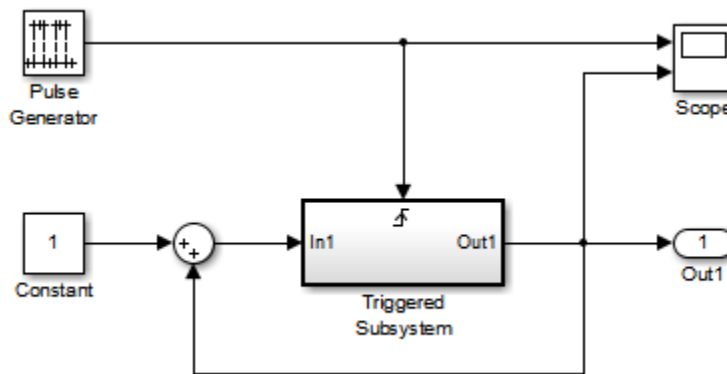


The code generator does not produce code that solves algebraic loops. This restriction includes models that use Algebraic Constraint blocks in feedback paths. However, the

Simulink engine can often eliminate algebraic loops that arise, by grouping equations in certain ways in models that contain them. It does this by separating the update and output functions to avoid circular dependencies. See “Algebraic Loops” in the Simulink documentation for details.

Algebraic Loops in Triggered Subsystems

While the Simulink engine can minimize algebraic loops involving atomic and enabled subsystems, a special consideration applies to some triggered subsystems. An example for which code can be generated is shown in the following model and triggered subsystem.



The default Simulink behavior is to combine output and update methods for the subsystem, which creates an apparent algebraic loop, even though the Unit Delay block in the subsystem has no direct feedthrough.

You can allow the Simulink engine to solve the problem by splitting the output and update methods of triggered and enabled-triggered subsystems when feasible. If you want the code generator to take advantage of this feature, select the **Minimize algebraic loop occurrences** check box in the Subsystem Parameters dialog box. Select this option to avoid algebraic loop warnings in triggered subsystems involved in loops.

Note: If you check this box, the generated code for the subsystem might contain split output and update methods, even if the subsystem is not actually involved in a loop. Also, if a direct feedthrough block (such as a Gain block) is connected to the inport in the

above triggered subsystem, the Simulink engine cannot solve the problem, and the code generator is unable to generate code.

A similar **Minimize algebraic loop occurrences** option appears on the **Model Referencing** pane of the Configuration Parameters dialog box. Selecting it enables the Simulink Coder software to generate code for models containing Model blocks that are involved in algebraic loops.

Modeling Guidelines for Blocks

Code generation modeling guidelines include recommended model settings, block usage, and block parameters. When you develop models for code generation, use these guidelines.

For more information, see “Modeling Guidelines” in the Simulink documentation.

Code Generation Modeling Guidelines	“cgsl_0101: Zero-based indexing” “cgsl_0102: Evenly spaced breakpoints in lookup tables” “cgsl_0103: Precalculated signals and parameters” “cgsl_0104: Modeling global shared memory using data stores” “cgsl_0105: Modeling local shared memory using data stores” “cgsl_0201: Redundant Unit Delay and Memory blocks”
-------------------------------------	--

See Also

“Modeling Guidelines for Subsystems” on page 2-37 | “Modeling Guidelines for Charts” on page 2-39 | “Modeling Guidelines for MATLAB Functions” on page 2-41 | “Modeling Guidelines for Model Configuration” on page 2-42

Modeling Guidelines for Subsystems

When you develop models and generate code for subsystems, use the modeling guideline recommendations.

For more information, see “Modeling Guidelines” in the Simulink documentation.

Code Generation Modeling Guidelines	“cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks”
High-Integrity Systems Modeling Guidelines	“hisl_0009: Usage of For Iterator Subsystem blocks” “hisl_0010: Usage of If blocks and If Action Subsystem blocks” “hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks” “hisl_0023: Verification of model and subsystem variants”
MathWorks Automotive Advisory Board (MAAB) Control Algorithm Guidelines	db_0040: Model hierarchy db_0042: Port block in Simulink models db_0081: Unconnected signals, block inputs and block outputs db_0143: Similar block types on the model levels db_0144: Use of Subsystems db_0146: Triggered, enabled, conditional Subsystems jc_0111: Direction of Subsystem jc_0201: Usable characters for Subsystem names jc_0231: Usable characters for block names jc_0281: Naming of Trigger Port block and Enable Port block jc_0321: Trigger layer jc_0331: Structure layer

	jc_0351: Methods of initialization
	jm_0002: Block resizing
	na_0005: Port block name visibility in Simulink models
	na_0006: Guidelines for mixed use of Simulink and Stateflow
	na_0008: Display of labels on signals
	na_0009: Entry versus propagation of signal labels
	na_0012: Use of Switch vs. If-Then-Else Action Subsystem

See Also

“Modeling Guidelines for Blocks” on page 2-36 | “Modeling Guidelines for Charts” on page 2-39 | “Modeling Guidelines for MATLAB Functions” on page 2-41 | “Modeling Guidelines for Model Configuration” on page 2-42

Modeling Guidelines for Charts

When you develop models and generate code for charts, use the modeling guideline recommendations.

For more information, see “Modeling Guidelines” in the Simulink documentation.

High-Integrity Systems Modeling Guidelines	<p>“hisf_0001: Mealy and Moore semantics”</p> <p>“hisf_0002: User-specified state/transition execution order”</p> <p>“hisf_0009: Strong data typing (Simulink and Stateflow boundary)”</p> <p>“hisf_0011: Stateflow debugging settings”</p> <p>“hisf_0003: Usage of bitwise operations”</p> <p>“hisf_0004: Usage of recursive behavior”</p> <p>“hisf_0007: Usage of junction conditions (maintaining mutual exclusion)”</p> <p>“hisf_0010: Usage of transition paths (looping out of parent of source and destination objects)”</p> <p>“hisf_0012: Chart comments”</p> <p>“hisf_0013: Usage of transition paths (crossing parallel state boundaries)”</p> <p>“hisf_0014: Usage of transition paths (passing through states)”</p> <p>“hisf_0015: Strong data typing (casting variables and parameters in expressions)”</p>
MathWorks Automotive Advisory Board (MAAB) Control Algorithm Guidelines	<p>db_0127: MATLAB commands in Stateflow</p> <p>db_0151: State machine patterns for transition actions</p> <p>jc_0451: Use of unary minus on unsigned integers in Stateflow</p> <p>jc_0481: Use of hard equality comparisons for floating point numbers in Stateflow</p>

jc_0501: Format of entries in a State block
jc_0511: Setting the return value from a graphical function
jc_0521: Use of the return value from graphical functions
jc_0531: Placement of the default transition
jc_0541: Use of tunable parameters in Stateflow
jm_0011: Pointers in Stateflow
jm_0012: Event broadcasts
na_0001: Bitwise Stateflow operators
na_0013: Comparison operation in Stateflow

See Also

“Modeling Guidelines for Blocks” on page 2-36 | “Modeling Guidelines for Subsystems” on page 2-37 | “Modeling Guidelines for MATLAB Functions” on page 2-41 | “Modeling Guidelines for Model Configuration” on page 2-42

Modeling Guidelines for MATLAB Functions

When you develop models and generate code for MATLAB Functions, use the modeling guideline recommendations.

For more information, see “Modeling Guidelines” in the Simulink documentation.

High-Integrity Systems Modeling Guidelines	“himl_0001: Usage of standardized MATLAB function headers”
	“himl_0002: Strong data typing at MATLAB function boundaries”
	“himl_0003: Limitation of MATLAB function complexity”
	“himl_0005: Usage of global variables in MATLAB functions”

See Also

“Modeling Guidelines for Blocks” on page 2-36 | “Modeling Guidelines for Subsystems” on page 2-37 | “Modeling Guidelines for Charts” on page 2-39 | “Modeling Guidelines for Model Configuration” on page 2-42

Modeling Guidelines for Model Configuration

When you develop models and generate code, use the modeling guideline configuration recommendations.

For more information, see “Modeling Guidelines” in the Simulink documentation.

Code Generation Modeling Guidelines	<p>“cgsl_0301: Prioritization of code generation objectives for code efficiency”</p> <p>“cgsl_0302: Diagnostic settings for multirate and multitasking models”</p>
High-Integrity Systems Modeling Guidelines	<p>“hisl_0043: Configuration Parameters > Diagnostics > Solver”</p> <p>“hisl_0044: Configuration Parameters > Diagnostics > Sample Time”</p> <p>“hisl_0301: Configuration Parameters > Diagnostics > Compatibility”</p> <p>“hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters”</p> <p>“hisl_0303: Configuration Parameters > Diagnostics > Merge block”</p> <p>“hisl_0304: Configuration Parameters > Diagnostics > Model initialization”</p> <p>“hisl_0305: Configuration Parameters > Diagnostics > Debugging”</p> <p>“hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals”</p> <p>“hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses”</p> <p>“hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls”</p> <p>“hisl_0309: Configuration Parameters > Diagnostics > Type Conversion”</p> <p>“hisl_0310: Configuration Parameters > Diagnostics > Model Referencing”</p>

“hisl_0311: Configuration Parameters > Diagnostics > Stateflow”

See Also

“Modeling Guidelines for Blocks” on page 2-36 | “Modeling Guidelines for Subsystems” on page 2-37 | “Modeling Guidelines for Charts” on page 2-39 | “Modeling Guidelines for MATLAB Functions” on page 2-41

Timers in Simulink Coder

- “Absolute and Elapsed Time Computation” on page 3-2
- “Access Timers Programmatically” on page 3-5
- “Generate Code for an Elapsed Time Counter” on page 3-9
- “Absolute Time Limitations” on page 3-12

Absolute and Elapsed Time Computation

In this section...

“About Timers” on page 3-2

“Timers for Periodic and Asynchronous Tasks” on page 3-3

“Allocation of Timers” on page 3-3

“Integer Timers in Generated Code” on page 3-3

“Elapsed Time Counters in Triggered Subsystems” on page 3-4

About Timers

Certain blocks require the value of either *absolute* time (that is, the time from the start of program execution to the present time) or *elapsed* time (for example, the time elapsed between two trigger events). Targets that support the real-time model (`rtModel`) data structure provide efficient time computation services to blocks that request absolute or elapsed time. Absolute and elapsed timer features include

- Timers are implemented as unsigned integers in generated code.
- In multirate models, at most one timer is allocated per rate. If no blocks executing at a given rate require a timer, a timer is not allocated to that rate. This minimizes memory allocated for timers and significantly reduces overhead involved in maintaining timers.
- Allocation of elapsed time counters for use of blocks within triggered subsystems is minimized, further reducing memory usage and overhead.
- S-function and TLC APIs let your S-functions access timers, in simulation and code generation.
- The word size of the timers is determined by a user-specified maximum counter value, **Application lifespan (days)**. If you specify this value, timers will not overflow. For more information, see “Control Memory Allocation for Time Counters” on page 31-15.

See “Absolute Time Limitations” on page 3-12 for more information about absolute time and the restrictions that it imposes.

Timers for Periodic and Asynchronous Tasks

Timing services provided for blocks execute within *periodic* tasks (that is, tasks running at the model base rate or subrates).

The code generator also provides timer support for blocks whose execution is *asynchronous* with respect to the periodic timing source of the model. See the following topics:

- “Timers in Asynchronous Tasks” on page 5-42
- “Create a Customized Asynchronous Library” on page 5-45

Allocation of Timers

If you create or maintain an S-Function block that requires absolute or elapsed time data, it must register the requirement (see “Access Timers Programmatically” on page 3-5). In multirate models, timers are allocated on a per-rate basis. For example, consider a model structured as follows:

- There are three rates, A, B, and C, in the model.
- No blocks running at rate B require absolute or elapsed time.
- Two blocks running at rate C register a requirement for absolute time.
- One block running at rate A registers a requirement for absolute time.

In this case, two timers are generated, running at rates A and C respectively. The timing engine updates the timers as the tasks associated with rates A and C execute. Blocks executing at rates A and C obtain time data from the timers associated with rates A and C.

Integer Timers in Generated Code

In the generated code, timers for absolute and elapsed time are implemented as unsigned integers. The default size is 64 bits. This is the amount of memory allocated for a timer if you specify a value of `inf` for the **Application lifespan (days)** parameter. For an application with a sample rate of 1000 MHz, a 64-bit counter will not overflow for more than 500 years. See “Timers in Asynchronous Tasks” on page 5-42 and “Control Memory Allocation for Time Counters” on page 31-15 for more information.

Elapsed Time Counters in Triggered Subsystems

Some blocks, such as the Discrete-Time Integrator block, perform computations requiring the elapsed time (delta T) since the previous block execution. Blocks requiring elapsed time data must register the requirement (see “Access Timers Programmatically” on page 3-5). A triggered subsystem then allocates and maintains a single elapsed time counter if required. This timer functions at the subsystem level, not at the individual block level. The timer is generated if the triggered subsystem (or a unconditionally executed subsystem within the triggered subsystem) contains one or more blocks requiring elapsed time data.

Note: If you are using simplified initialization mode, elapsed time is reset on first execution after becoming enabled, whether or not the subsystem is configured to reset on enable. For more information, see “Underspecified initialization detection” in the Simulink documentation.

More About

- “Access Timers Programmatically” on page 3-5
- “Generate Code for an Elapsed Time Counter” on page 3-9
- “Optimize Memory Usage for Time Counters” on page 31-83
- “Absolute Time Limitations” on page 3-12

Access Timers Programmatically

In this section...

“About Timer APIs” on page 3-5

“C API for S-Functions” on page 3-5

“TLC API for Code Generation” on page 3-7

About Timer APIs

This topic describes APIs that let your S-functions take advantage of the efficiencies offered by absolute and elapsed timers. `SimStruct` macros are provided for use in simulation, and TLC functions are provided for inlined code generation. Note that

- To generate and use the new timers as described above, your S-functions must register the need to use an absolute or elapsed timer by calling `ssSetNeedAbsoluteTime` or `ssSetNeedElapseTime` in `mdlInitializeSampleTime`.
- Existing S-functions that read absolute time but do not register by using these macros continue to operate as expected, but generate less efficient code.

C API for S-Functions

The `SimStruct` macros described in this topic provide access to absolute and elapsed timers for S-functions during simulation.

In the functions below, the `SimStruct *S` argument is a pointer to the `simstruct` of the calling S-function.

- `void ssSetNeedAbsoluteTime(SimStruct *S, boolean b)`: if `b` is `TRUE`, registers that the calling S-function requires absolute time data, and allocates an absolute time counter for the rate at which the S-function executes (if such a counter has not already been allocated).
- `int ssGetNeedAbsoluteTime(SimStruct *S)`: returns 1 if the S-function has registered that it requires absolute time.
- `double ssGetTaskTime(SimStruct *S, tid)`: read absolute time for a given task with task identifier `tid`. `ssGetTaskTime` operates transparently, regardless of whether or not you use the new timer features. `ssGetTaskTime` is documented in the `SimStruct Functions` chapter of the Simulink documentation.

- `void ssSetNeedElapseTime(SimStruct *S, boolean b)`: if `b` is `TRUE`, registers that the calling S-function requires elapsed time data, and allocates an elapsed time counter for the triggered subsystem in which the S-function executes (if such a counter has not already been allocated). See also “Elapsed Time Counters in Triggered Subsystems” on page 3-4.
- `int ssGetNeedElapseTime(SimStruct *S)`: returns 1 if the S-function has registered that it requires elapsed time.
- `void ssGetElapseTime(SimStruct *S, (double *)elapseTime)`: returns, to the location pointed to by `elapseTime`, the value (as a `double`) of the elapsed time counter associated with the S-function.
- `void ssGetElapseTimeCounterDtype(SimStruct *S, (int *)dtype)`: returns the data type of the elapsed time counter associated with the S-function to the location pointed to by `dtype`. This function is intended for use with the `ssGetElapseTimeCounter` function (see below).
- `void ssGetElapseResolution(SimStruct *S, (double *)resolution)`: returns the resolution (that is, the sample time) of the elapsed time counter associated with the S-function to the location pointed to by `resolution`. This function is intended for use with the `ssGetElapseTimeCounter` function (see below).
- `void ssGetElapseTimeCounter(SimStruct *S, (void *)elapseTime)`: This function is provided for the use of blocks that require the elapsed time values for fixed-point computations. `ssGetElapseTimeCounter` returns, to the location pointed to by `elapseTime`, the integer value of the elapsed time counter associated with the S-function. If the counter size is 64 bits, the value is returned as an array of two 32-bit words, with the low-order word stored at the lower address.

To determine how to access the returned counter value, obtain the data type of the counter by calling `ssGetElapseTimeCounterDtype`, as in the following code:

```
int    *y_dtype;
ssGetElapseTimeCounterDtype(S, y_dtype);

switch(*y_dtype) {
  case SS_DOUBLE_UINT32:
    {
      uint32_T dataPtr[2];
      ssGetElapseTimeCounter(S, dataPtr);
    }
    break;
  case SS_UINT32:
```

```

        {
            uint32_T dataPtr[1];
            ssGetElapseTimeCounter(S, dataPtr);
        }
        break;
case SS_UINT16:
    {
        uint16_T dataPtr[1];
        ssGetElapseTimeCounter(S, dataPtr);
    }
    break;
case SS_UINT8:
    {
        uint8_T dataPtr[1];
        ssGetElapseTimeCounter(S, dataPtr);
    }
    break;
case SS_DOUBLE:
    {
        real_T dataPtr[1];
        ssGetElapseTimeCounter(S, dataPtr);
    }
    break;
default:
    ssSetErrorStatus(S, "Invalid data type for elapse time
        counter");
    break;
}

```

If you want to use the actual elapsed time, issue a call to the `ssGetElapseTime` function to access the elapsed time directly. You do not need to get the counter value and then calculate the elapsed time.

```

double *y_elapseTime;
.
.
.
ssGetElapseTime(S, elapseTime)

```

TLC API for Code Generation

The following TLC functions support elapsed time counters in generated code when you inline S-functions by writing TLC scripts for them.

- `LibGetTaskTimeFromTID(block)`: Generates code to read the absolute time for the task in which `block` executes.

`LibGetTaskTimeFromTID` is documented with other sample time functions in the TLC Function Library Reference pages of the Target Language Compiler documentation.

Note Do not use `LibGetT` for this purpose. `LibGetT` always reads the base rate (`tid 0`) timer. If `LibGetT` is called for a block executing at a subrate, the wrong timer is read, causing serious errors.

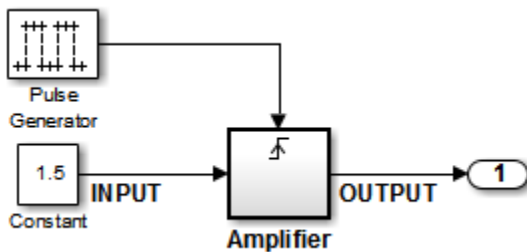
- `LibGetElapseTime(system)`: Generates code to read the elapsed time counter for `system`. (`system` is the parent system of the calling block.) See “Generate Code for an Elapsed Time Counter” on page 3-9 for an example of code generated by this function.
- `LibGetElapseTimeCounter(system)`: Generates code to read the integer value of the elapsed time counter for `system`. (`system` is the parent system of the calling block.) This function should be used in conjunction with `LibGetElapseTimeCounterDtypeId` and `LibGetElapseTimeResolution`. (See the discussion of `ssGetElapseTimeCounter` above.)
- `LibGetElapseTimeCounterDtypeId(system)`: Generates code that returns the data type of the elapsed time counter for `system`. (`system` is the parent system of the calling block.)
- `LibGetElapseTimeResolution(system)`: Generates code that returns the resolution of the elapsed time counter for `system`. (`system` is the parent system of the calling block.)

More About

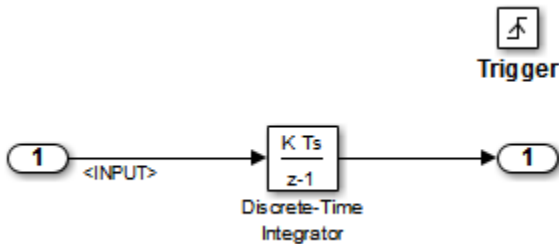
- “Absolute and Elapsed Time Computation” on page 3-2
- “Generate Code for an Elapsed Time Counter” on page 3-9
- “Absolute Time Limitations” on page 3-12

Generate Code for an Elapsed Time Counter

This example shows a model that includes a triggered subsystem, **Amplifier**, consisting of a Discrete-Time Integrator block that uses an elapsed time counter. The model `ex_elapseTime` is in the folder `matlab/help/toolbox/rtw/examples`.



ex_elapseTime Model



Amplifier Subsystem

Code in the generated header file `ex_elapseTime.h` for the model uses 64 bits to implement the timer for the base rate (`clockTick0` and `clockTickH0`).

```

/*
 * Timing:
 * The following substructure contains information regarding
 * the timing information for the model.
 */
struct {
    time_T taskTime0;
    uint32_T clockTick0;

```

```

uint32_T clockTickH0;
time_T stepSize0;
time_T tFinal;
boolean_T stopRequestedFlag;
} Timing;

```

The code generator allocates storage for the previous-time value and elapsed-time value of the Amplifier subsystem (Amplifier_PREV_T) in the D_Work(states) structure in ex_elapsedTime.h.

```

/* Block states (auto storage) for system '<Root>' */
typedef struct {
    real_T DiscreteTimeIntegrator_DSTATE; /* '<S1>/Discrete-Time Integrator' */
    int32_T clockTickCounter;             /* '<Root>/Pulse Generator' */
    uint32_T Amplifier_ELAPS_T[2];        /* '<Root>/Amplifier' */
    uint32_T Amplifier_PREV_T[2];         /* '<Root>/Amplifier' */
} DW_ex_elapseTime_T;

```

The elapsed time computation is performed as follows within the ex_elapseTime_step function:

```

/* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
 * TriggerPort: '<S1>/Trigger'
 */
zcEvent = rt_ZCFcn(RISING_ZERO_CROSSING,
    &ex_elapseTime_PrevZCX.Amplifier_Trig_ZCE,
    ((real_T)rtb_PulseGenerator));
if (zcEvent != NO_ZCEVENT) {
    elapseT_H = ex_elapseTime_M->Timing.clockTickH0 -
        ex_elapseTime_DW.Amplifier_PREV_T[1];
    if (ex_elapseTime_DW.Amplifier_PREV_T[0] >
        ex_elapseTime_M->Timing.clockTick0) {
        elapseT_H--;
    }

    ex_elapseTime_DW.Amplifier_ELAPS_T[0] = ex_elapseTime_M->Timing.clockTick0 -
        ex_elapseTime_DW.Amplifier_PREV_T[0];
    ex_elapseTime_DW.Amplifier_PREV_T[0] = ex_elapseTime_M->Timing.clockTick0;
    ex_elapseTime_DW.Amplifier_ELAPS_T[1] = elapseT_H;
    ex_elapseTime_DW.Amplifier_PREV_T[1] = ex_elapseTime_M->Timing.clockTickH0;
}

```

As shown above, the elapsed time is maintained as a state of the triggered subsystem. The Discrete-Time Integrator block finally performs its output and update computations using the elapsed time.

```
/* DiscreteIntegrator: '<S1>/Discrete-Time Integrator' */  
OUTPUT = ex_elapseTime_DW.DiscreteTimeIntegrator_DSTATE;  
  
/* Update for DiscreteIntegrator: '<S1>/Discrete-Time Integrator' incorporates:  
 * Constant: '<Root>/Constant'  
 */  
ex_elapseTime_DW.DiscreteTimeIntegrator_DSTATE += 0.3 * (real_T)  
ex_elapseTime_DW.Amplifier_ELAPS_T[0] * 1.5;
```

More About

- “Absolute and Elapsed Time Computation” on page 3-2
- “Absolute Time Limitations” on page 3-12

Absolute Time Limitations

Absolute time is the time that has elapsed from the beginning of program execution to the present time, as distinct from *elapsed time*, the interval between two events. See “Absolute and Elapsed Time Computation” on page 3-2 for more information.

When you design an application that is intended to run indefinitely, you must take care when logging time values, or using charts or blocks that depend on absolute time. If the value of time reaches the largest value that can be represented by the data type used by the timer to store time, the timer overflows and the logged time or block output is incorrect.

If your target uses `rtModel`, you can avoid timer overflow by specifying a value for the **Application life span** parameter. See “Integer Timers in Generated Code” on page 3-3 for more information.

The following limitations apply to absolute time:

- If you log time values by opening the Configuration Parameters dialog box and enabling **Data Import/Export > Time** parameter, your model uses absolute time.
- Every Stateflow chart that uses time is dependent on absolute time. The only way to eliminate the dependency is to change the Stateflow chart to not use time.
- The following Simulink blocks depend on absolute time:
 - Backlash
 - Chirp Signal
 - Clock
 - Derivative
 - Digital Clock
 - Discrete-Time Integrator (only when used in triggered subsystems)
 - From File
 - From Workspace
 - Pulse Generator
 - Ramp
 - Rate Limiter
 - Repeating Sequence

- Signal Generator
- Sine Wave (only when the **Sine type** parameter is set to **Time-based**)
- Step
- To File
- To Workspace (only when logging to **StructureWithTime** format)
- Transport Delay
- Variable Time Delay
- Variable Transport Delay

In addition to the Simulink blocks above, blocks in other blocksets may depend on absolute time. See the documentation for the blocksets that you use.

More About

- “Absolute and Elapsed Time Computation” on page 3-2

Time-Based Scheduling in Simulink Coder

Time-Based Scheduling and Code Generation

In this section...
“Sample Time Considerations” on page 4-2
“Tasking Modes” on page 4-2
“Model Execution and Rate Transitions” on page 4-4
“Execution During Simulink Model Simulation” on page 4-5
“Model Execution in Real Time” on page 4-5
“Single-Tasking Versus Multitasking Operation” on page 4-6

Sample Time Considerations

Simulink models run at one or more sample times. The Simulink product provides considerable flexibility in building multirate systems, that is, systems with more than one sample time. However, this same flexibility also allows you to construct models for which the code generator cannot generate real-time code for execution in a multitasking environment. To make multirate models operate as expected in real time (that is, to give the right answers), you sometimes must modify your model or instruct the Simulink engine to modify the model for you. In general, the modifications involve placing Rate Transition blocks between blocks that have unequal sample times. The following sections discuss issues you must address to use a multirate model in a multitasking environment. For a comprehensive discussion of sample times, including rate transitions, see “What Is Sample Time?”, “Sample Times in Subsystems”, “Sample Times in Systems”, “Resolve Rate Transitions”, and associated topics.

Tasking Modes

There are two execution modes for a fixed-step Simulink model: single-tasking and multitasking. These modes are available only for fixed-step solvers. To select an execution mode, use the **Treat each discrete rate as a separate task** checkbox on the **Solver** pane of the Configuration Parameters dialog box. When this parameter is selected, multitasking execution is applied for a multirate model. When this option is cleared, single-tasking execution is applied.

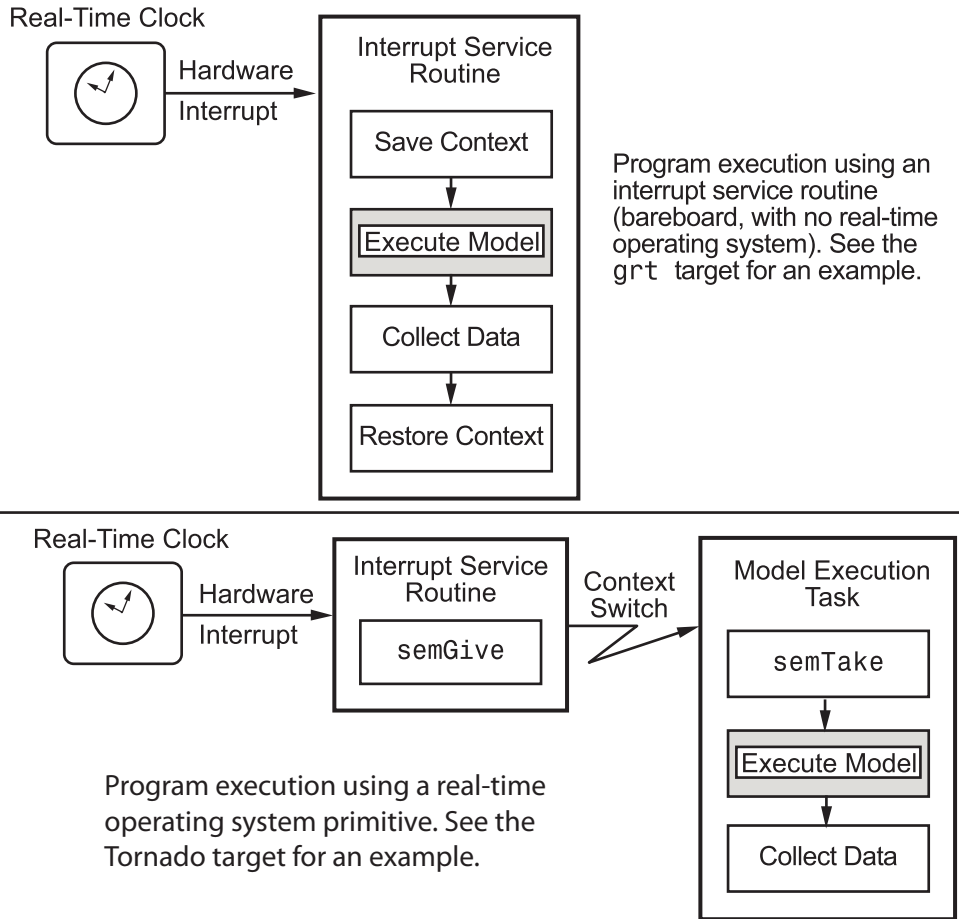
Note: A model that is multirate and uses multitasking cannot reference a multirate model that uses single-tasking.

Execution of models in a real-time system can be done with the aid of a real-time operating system, or it can be done on *bare-metal* target hardware, where the model runs in the context of an interrupt service routine (ISR).

The fact that a system (such as The Open Group UNIX[®] or Microsoft[®] Windows systems) is multitasking does not imply that your program can execute in real time. This is because the program might not preempt other processes when required.

In operating systems (such as PC-DOS) where only one process can exist at a given time, an interrupt service routine (ISR) must perform the steps of saving the processor context, executing the model code, collecting data, and restoring the processor context.

Other operating systems, such as POSIX-compliant ones, provide automatic context switching and task scheduling. This simplifies the operations performed by the ISR. In this case, the ISR simply enables the model execution task, which is normally blocked. The next figure illustrates this difference.



Model Execution and Rate Transitions

To generate code that executes as expected in real time, you (or the Simulink engine) might need to identify and handle sample rate transitions within the model. In multitasking mode, by default the Simulink engine flags errors during simulation if the model contains invalid rate transitions, although you can use the **Multitask rate transition** diagnostic to alter this behavior. A similar diagnostic, called **Single task rate transition**, exists for single-tasking mode.

To avoid raising rate transition errors, insert Rate Transition blocks between tasks. You can request that the Simulink engine handle rate transitions automatically by inserting hidden Rate Transition blocks. See “Automatic Rate Transition” on page 4-25 for an explanation of this option.

To understand such problems, first consider how Simulink simulations differ from real-time programs.

Execution During Simulink Model Simulation

Before the Simulink engine simulates a model, it orders the blocks based upon their topological dependencies. This includes expanding virtual subsystems into the individual blocks they contain and flattening the entire model into a single list. Once this step is complete, each block is executed in order.

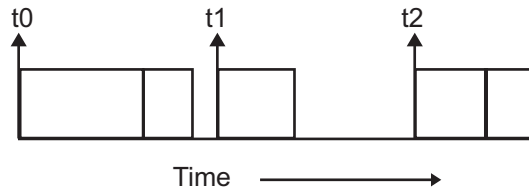
The key to this process is the ordering of blocks. A block whose output is directly dependent on its input (that is, a block with direct feedthrough) cannot execute until the block driving its input executes.

Some blocks set their outputs based on values acquired in a previous time step or from initial conditions specified as a block parameter. The output of such a block is determined by a value stored in memory, which can be updated independently of its input. During simulation, computations are performed prior to advancing the variable corresponding to time. This results in computations occurring instantaneously (that is, no computational delay).

Model Execution in Real Time

A real-time program differs from a Simulink simulation in that the program must execute the model code synchronously with real time. Every calculation results in some computational delay. This means the sample intervals cannot be shortened or lengthened (as they can be in a Simulink simulation), which leads to less efficient execution.

Consider the following timing figure.



Note the processing inefficiency in the sample interval t_1 . That interval cannot be compressed to increase execution speed because, by definition, sample times are clocked in real time.

You can circumvent this potential inefficiency by using the multitasking mode. The multitasking mode defines tasks with different priorities to execute parts of the model code that have different sample rates.

See “Multitasking and Pseudomultitasking Modes” on page 4-12 for a description of how this works. It is important to understand that section before proceeding here.

Single-Tasking Versus Multitasking Operation

Single-tasking programs require longer sample intervals, because all computations must be executed within each clock period. This can result in inefficient use of available CPU time, as shown in the previous figure.

Multitasking mode can improve the efficiency of your program if the model is large and has many blocks executing at each rate.

However, if your model is dominated by a single rate, and only a few blocks execute at a slower rate, multitasking can actually degrade performance. In such a model, the overhead incurred in task switching can be greater than the time required to execute the slower blocks. In this case, it is more efficient to execute all blocks at the dominant rate.

If you have a model that can benefit from multitasking execution, you might need to modify your model by adding Rate Transition blocks (or instruct the Simulink engine to do so) to generate expected results.

For more information about the two modes of execution and examples, see “Modeling for Single-Tasking Execution” on page 4-8 and “Modeling for Multitasking Execution” on page 4-12.

More About

- “What Is Sample Time?”
- “Sample Times in Subsystems”
- “Sample Times in Systems”
- “Configure Time-Based Scheduling” on page 4-34
- “Sample Times in Subsystems”
- “Sample Times in Systems”
- “Resolve Rate Transitions”
- “Handle Rate Transitions” on page 4-20
- “Time-Based Scheduling and Code Generation” on page 4-2
- “Modeling for Single-Tasking Execution” on page 4-8
- “Modeling for Multitasking Execution” on page 4-12
- “Time-Based Scheduling Example Models” on page 4-36

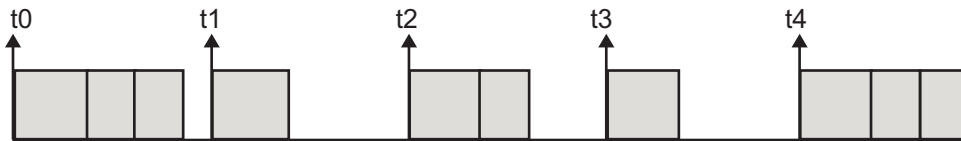
Modeling for Single-Tasking Execution

Single-Tasking Mode

You can execute model code in a strictly single-tasking manner. While this mode is less efficient with regard to execution speed, in certain situations, it can simplify your model.

In single-tasking mode, the base sample rate must define a time interval that is long enough to allow the execution of all blocks within that interval.

The next figure illustrates the inefficiency inherent in single-tasking execution.



Single-tasking system execution requires a base sample rate that is long enough to execute one step through the entire model.

Build a Program for Single-Tasking Execution

To use single-tasking execution, clear the **Treat each discrete rate as a separate task** checkbox on the **Solver** pane of the Configuration Parameters dialog box. If you select the checkbox, single-tasking mode is used in the following cases:

- If your model contains one sample time
- If your model contains a continuous and a discrete sample time and the fixed step size is equal to the discrete sample time

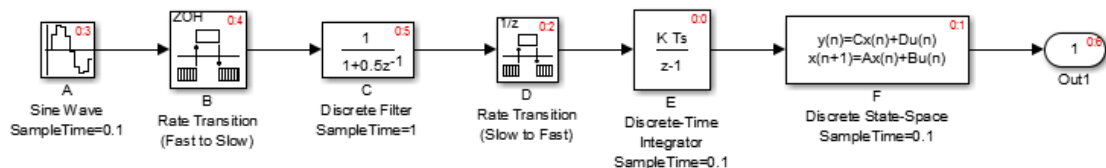
Single-Tasking Execution

This example examines how a simple multirate model executes in both real time and simulation, using a fixed-step solver. It considers operation in both single-tasking and multitasking modes, as determined by setting of the **Treat each discrete rate as a separate task** parameter on the **Solver** pane.

The example model is shown in the next figure. The discussion refers to the six blocks of the model as A through F, as labeled in the block diagram.

The execution order of the blocks (indicated in the upper right of each block) has been forced into the order shown by assigning higher priorities to blocks F, E, and D. The ordering shown is one possible valid execution ordering for this model. (See “Simulation Phases in Dynamic Systems” in the Simulink documentation.)

The execution order is determined by data dependencies between blocks. In a real-time system, the execution order determines the order in which blocks execute within a given time interval or task. This discussion treats the model's execution order as a given, because it is concerned with the allocation of block computations to tasks, and the scheduling of task execution.



Note The discussion and timing diagrams in this section are based on the assumption that the Rate Transition blocks are used in the default (protected/deterministic) mode, with the **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** options on.

This example considers the execution of the above model when the **Treat each discrete rate as a separate task** checkbox is cleared, which indicates the single-tasking mode.

In a single-tasking system, if the **Block reduction** option on the **All Parameters** tab is on, fast-to-slow Rate Transition blocks are optimized out of the model. The default case is shown (**Block reduction** on), so block B does not appear in the timing diagrams in this section. For more information, see “Block reduction”.

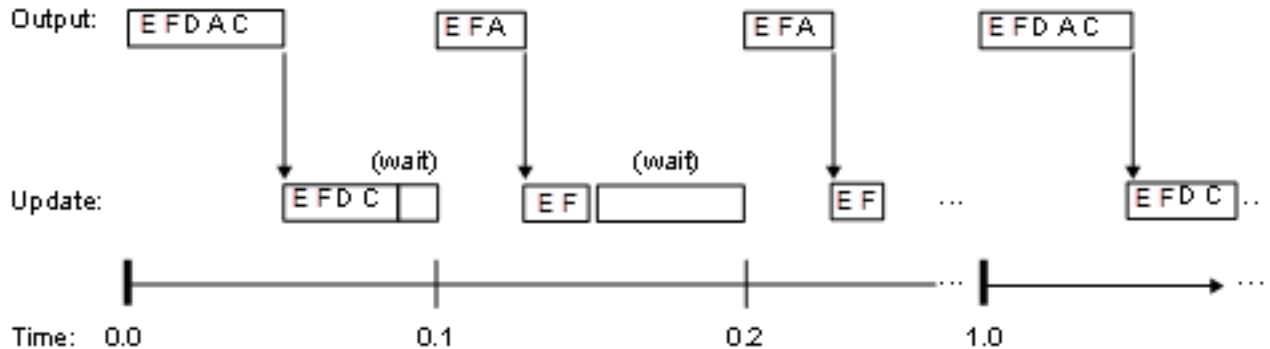
The following table shows, for each block in the model, the execution order, sample time, and whether the block has an output or update computation. Block A does not have discrete states, and accordingly does not have an update computation.

Execution Order and Sample Times (Single-Tasking)

Blocks (in Execution Order)	Sample Time (in Seconds)	Output	Update
E	0.1	Y	Y
F	0.1	Y	Y
D	1	Y	Y
A	0.1	Y	N
C	1	Y	Y

Real-Time Single-Tasking Execution

The next figure shows the scheduling of computations when the generated code is deployed in a real-time system. The generated program is shown running in real time, under control of interrupts from a 10 Hz timer.



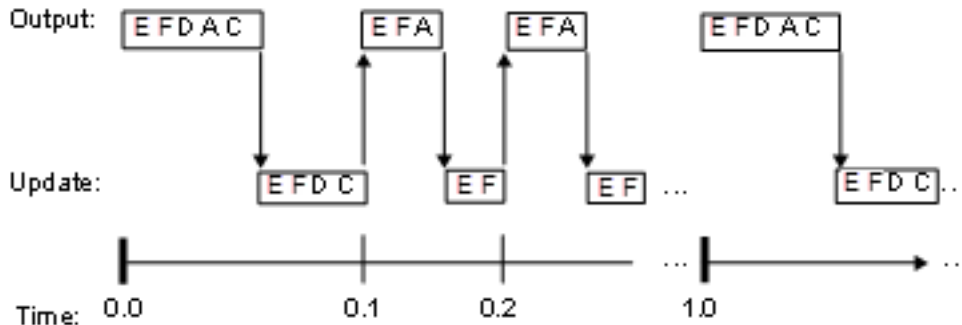
At time 0.0, 1.0, and every second thereafter, both the slow and fast blocks execute their output computations; this is followed by update computations for blocks that have states. Within a given time interval, output and update computations are sequenced in block execution order.

The fast blocks execute on every tick, at intervals of 0.1 second. Output computations are followed by update computations.

The system spends some portion of each time interval (labeled “wait”) idling. During the intervals when only the fast blocks execute, a larger portion of the interval is spent idling. This illustrates an inherent inefficiency of single-tasking mode.

Simulated Single-Tasking Execution

The next figure shows the execution of the model during the Simulink simulation loop.



Because time is simulated, the placement of ticks represents the iterations of the simulation loop. Blocks execute in exactly the same order as in the previous figure, but without the constraint of a real-time clock. Therefore there is no idle time between simulated sample periods.

More About

- “Time-Based Scheduling and Code Generation” on page 4-2
- “Configure Time-Based Scheduling” on page 4-34
- “Time-Based Scheduling Example Models” on page 4-36

Modeling for Multitasking Execution

Multitasking and Pseudomultitasking Modes

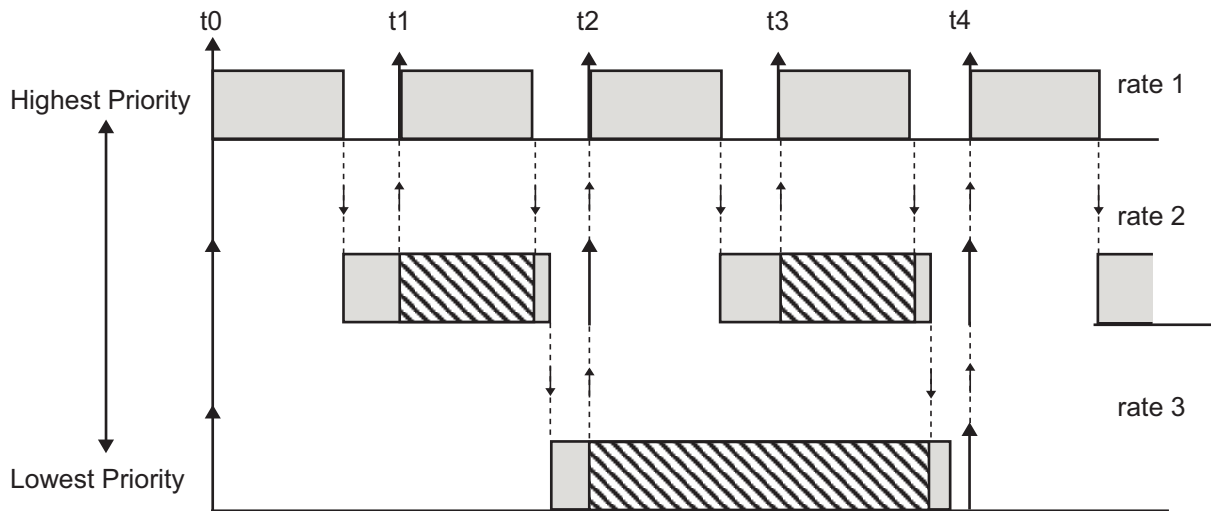
When periodic tasks execute in a multitasking mode, by default the blocks with the fastest sample rates are executed by the task with the highest priority, the next fastest blocks are executed by a task with the next higher priority, and so on. Time available in between the processing of high-priority tasks is used for processing lower priority tasks. This results in efficient program execution.

Where tasks are asynchronous rather than periodic, there may not necessarily be a relationship between sample rates and task priorities; the task with the highest priority need not have the fastest sample rate. You specify asynchronous task priorities using Async Interrupt and Task Sync blocks. You can switch the sense of what priority numbers mean by selecting or deselecting the Solver option **Higher priority value indicates higher task priority**.

In multitasking environments (that is, under a real-time operating system), you can define separate tasks and assign them priorities. For bare-metal target hardware (that is, no real-time operating system present), you cannot create separate tasks. However, generated application modules implement what is effectively a multitasking execution scheme using overlapped interrupts, accompanied by programmatic context switching.

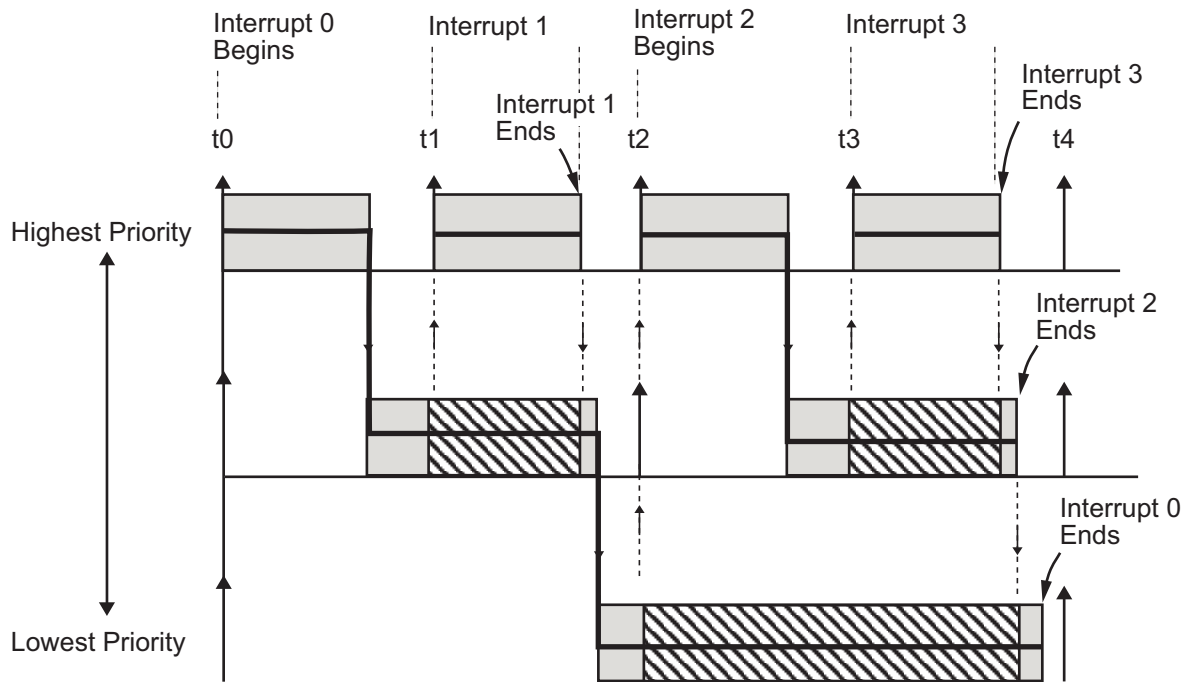
This means an interrupt can occur while another interrupt is currently in progress. When this happens, the current interrupt is preempted, the floating-point unit (FPU) context is saved, and the higher priority interrupt executes its higher priority (that is, faster sample rate) code. Once complete, control is returned to the preempted ISR.

The next figures illustrate how timing of tasks in multirate systems are handled by the code generator in multitasking, pseudomultitasking, and single-tasking environments.



- ↑ Vertical arrows indicate sample time hits.
- ⋮ Dotted lines with downward pointing arrows indicate the release of control to a lower priority task.
- ⋮ Dotted lines with upward pointing arrows indicate preemption by a higher priority task.
- Dark gray areas indicate task execution.
- ▨ Hashed areas indicate task preemption by a higher priority task.
- Light gray areas indicate task execution is pending.

The next figure shows how overlapped interrupts are used to implement pseudomultitasking. In this case, Interrupt 0 does not return until after Interrupts 1, 2, and 3.



Build a Program for Multitasking Execution

To use multitasking execution, select the **Treat each discrete rate as a separate task** check box on the **Solver** pane of the Configuration Parameters dialog box. This menu is active only if you select **Fixed-step** as the solver type. **Auto** mode results in a multitasking environment if your model has two or more different sample times. A model with a continuous and a discrete sample time runs in single-tasking mode if the fixed-step size is equal to the discrete sample time.

Execute Multitasking Models

In cases where the continuous part of a model executes at a rate that is different from the discrete part, or a model has blocks with different sample rates, the Simulink engine assigns each block a *task identifier* (τid) to associate the block with the task that executes at the block's sample rate.

You set sample rates and their constraints on the **Solver** pane of the Configuration Parameters dialog box. To generate code, select **Fixed-step** for the solver type. Certain restrictions apply to the sample rates that you can use:

- The sample rate of a block must be an integer multiple of the base (that is, the fastest) sample period.
- When **Periodic sample time constraint** is unconstrained, the base sample period is determined by the **Fixed step size** specified on the **Solvers** pane of the Configuration parameters dialog box.
- When **Periodic sample time constraint** is Specified, the base rate fixed-step size is the first element of the sample time matrix that you specify in the companion option **Sample time properties**. The **Solver** pane from the example model `rtwdemo_mrmtbb` shows an example.

Simulation time

Start time: Stop time:

Solver options

Type: Solver:

▼ Additional options

Fixed-step size (fundamental sample time):

Tasking and sample time options

Periodic sample time constraint:

Sample time properties:

Treat each discrete rate as a separate task

Automatically handle rate transition for data transfer

Higher priority value indicates higher task priority

- Continuous blocks execute by using an integration algorithm that runs at the base sample rate. The base sample period is the greatest common denominator of all rates in the model only when **Periodic sample time constraint** is set to **Unconstrained** and **Fixed step size** is **Auto**.

- The continuous and discrete parts of the model can execute at different rates only if the discrete part is executed at the same or a slower rate than the continuous part and is an integer multiple of the base sample rate.

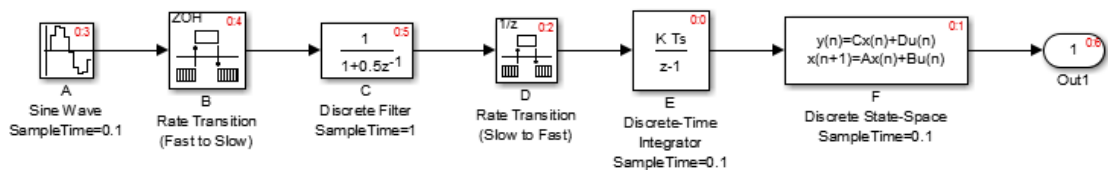
Multitasking Execution

This example examines how a simple multirate model executes in both real time and simulation, using a fixed-step solver. It considers operation in both single-tasking and multitasking modes, as determined by setting of the **Treat each discrete rate as a separate task** parameter on the **Solver** pane.

The example model is shown in the next figure. The discussion refers to the six blocks of the model as A through F, as labeled in the block diagram.

The execution order of the blocks (indicated in the upper right of each block) has been forced into the order shown by assigning higher priorities to blocks F, E, and D. The ordering shown is one possible valid execution ordering for this model. (See “Simulation Phases in Dynamic Systems” in the Simulink documentation.)

The execution order is determined by data dependencies between blocks. In a real-time system, the execution order determines the order in which blocks execute within a given time interval or task. This discussion treats the model's execution order as a given, because it is concerned with the allocation of block computations to tasks, and the scheduling of task execution.



Note The discussion and timing diagrams in this section are based on the assumption that the Rate Transition blocks are used in the default (protected/deterministic) mode, with the **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** options on.

This example considers the execution of the above model when the solver **Tasking mode** is **MultiTasking**. Block computations are executed under two tasks, prioritized by rate:

- The slower task, which gets the lower priority, is scheduled to run every second. This is called the *1 second task*.
- The faster task, which gets higher priority, is scheduled to run 10 times per second. This is called the *0.1 second task*. The 0.1 second task can preempt the 1 second task.

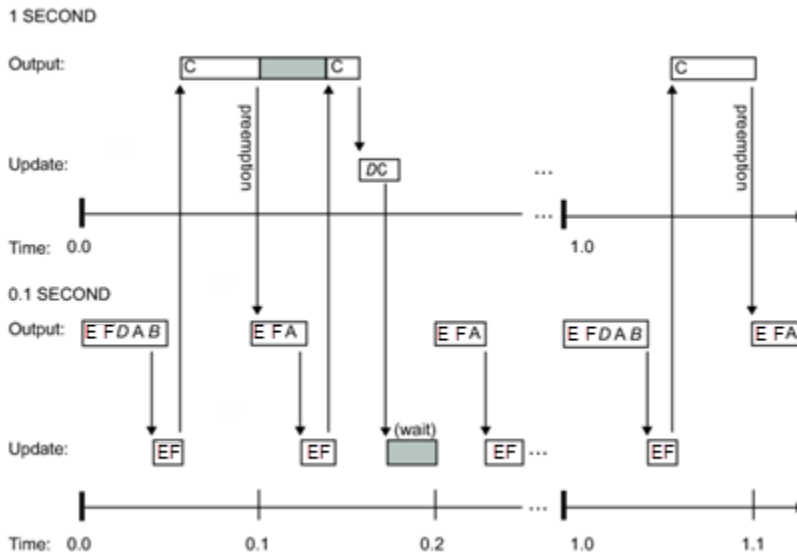
The following table shows, for each block in the model, the execution order, the task under which the block runs, and whether the block has an output or update computation. Blocks A and B do not have discrete states, and accordingly do not have an update computation.

Task Allocation of Blocks in Multitasking Execution

Blocks (in Execution Order)	Task	Output	Update
E	0.1 second task	Y	Y
F	0.1 second task	Y	Y
D	The Rate Transition block uses port-based sample times. Output runs at the output port sample time under 0.1 second task. Update runs at input port sample time under 1 second task. For more information on port-based sample times, see “Sample Times for Model Referencing” in the Simulink documentation.	Y	Y
A	0.1 second task	Y	N
B	The Rate Transition block uses port-based sample times. Output runs at the output port sample time under 0.1 second task. For more information on port-based sample times, see “Sample Times for Model Referencing” in the Simulink documentation.	Y	N
C	1 second task	Y	Y

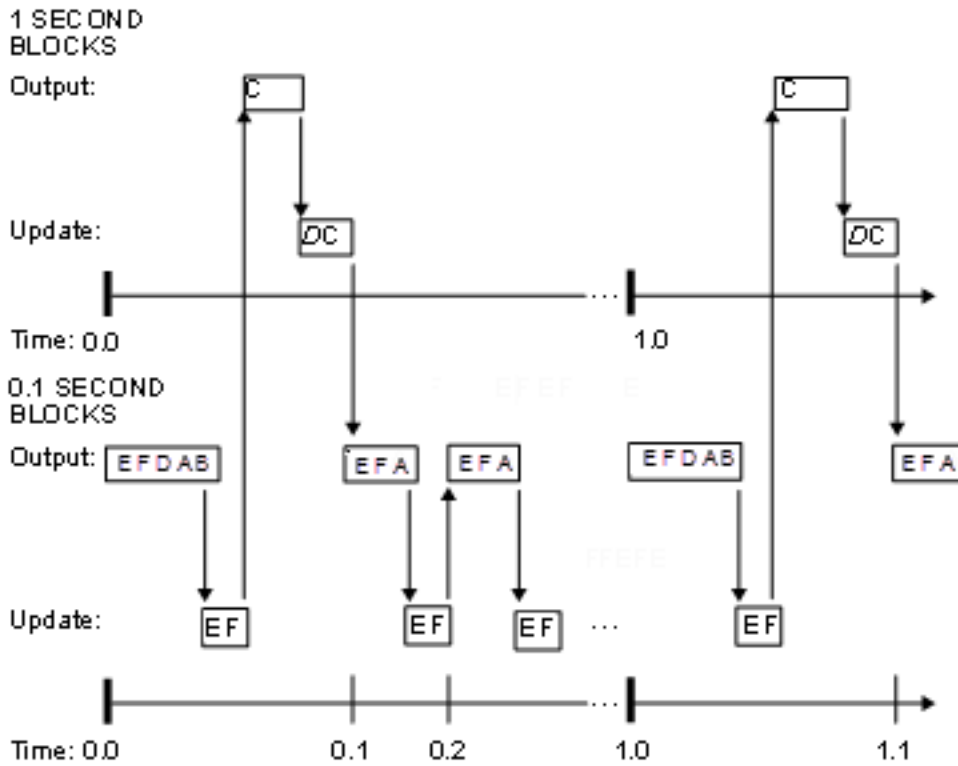
Real-Time Multitasking Execution

The next figure shows the scheduling of computations in `MultiTasking` solver mode when the generated code is deployed in a real-time system. The generated program is shown running in real time, as two tasks under control of interrupts from a 10 Hz timer.



Simulated Multitasking Execution

The next figure shows the Simulink execution of the same model, in `MultiTasking` solver mode. In this case, the Simulink engine runs the blocks in one thread of execution, simulating multitasking. No preemption occurs.



More About

- “Time-Based Scheduling and Code Generation” on page 4-2
- “Sample Times in Subsystems”
- “Sample Times in Systems”
- “Configure Time-Based Scheduling” on page 4-34
- “Resolve Rate Transitions”
- “Handle Rate Transitions” on page 4-20
- “Time-Based Scheduling Example Models” on page 4-36

Handle Rate Transitions

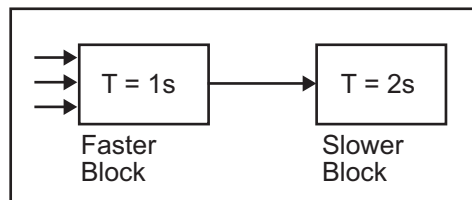
Rate Transitions

Two periodic sample rate transitions can exist within a model:

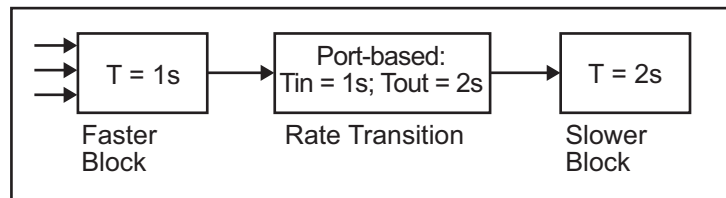
- A faster block driving a slower block
- A slower block driving a faster block

The following sections concern models with periodic sample times with zero offset only. Other considerations apply to multirate models that involve asynchronous tasks. For details on how to generate code for asynchronous multitasking, see “Asynchronous Support” on page 5-2.

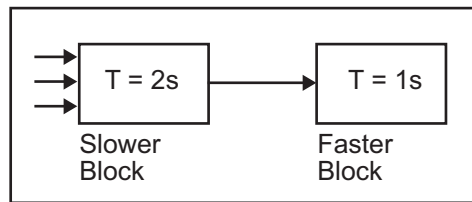
In multitasking and pseudomultitasking systems, differing sample rates can cause blocks to be executed in the wrong order. To prevent possible errors in calculated data, you must control model execution at these transitions. When connecting faster and slower blocks, you or the Simulink engine must add Rate Transition blocks between them. Fast-to-slow transitions are illustrated in the next figure.



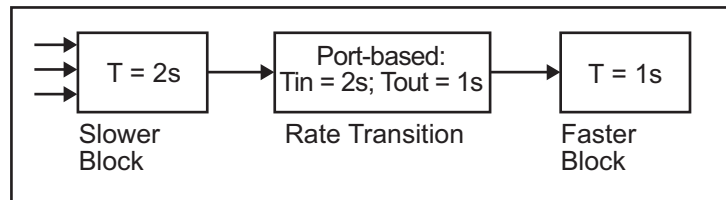
becomes



Slow-to-fast transitions are illustrated in the next figure.



becomes



Note: Although the Rate Transition block offers a superset of the capabilities of the Unit Delay block (for slow-to-fast transitions) and the Zero-Order Hold block (for fast-to-slow transitions), you should use the Rate Transition block instead of these blocks.

Data Transfer Problems

Rate Transition blocks deal with issues of data integrity and determinism associated with data transfer between blocks running at different rates.

- *Data integrity:* A problem of data integrity exists when the input to a block changes during the execution of that block. Data integrity problems can be caused by preemption.

Consider the following scenario:

- A faster block supplies the input to a slower block.
- The slower block reads an input value V_1 from the faster block and begins computations using that value.
- The computations are preempted by another execution of the faster block, which computes a new output value V_2 .
- A data integrity problem now arises: when the slower block resumes execution, it continues its computations, now using the “new” input value V_2 .

Such a data transfer is called *unprotected*. “Faster to Slower Transitions in Real Time” on page 4-29 shows an unprotected data transfer.

In a *protected* data transfer, the output V_1 of the faster block is held until the slower block finishes executing.

- *Deterministic* versus *nondeterministic* data transfer: In a *deterministic* data transfer, the timing of the data transfer is completely predictable, as determined by the sample rates of the blocks.

The timing of a *nondeterministic* data transfer depends on the availability of data, the sample rates of the blocks, and the time at which the receiving block begins to execute relative to the driving block.

You can use the Rate Transition block to protect data transfers in your application and make them deterministic. These characteristics are considered desirable in most applications. However, the Rate Transition block supports flexible options that allow you to compromise data integrity and determinism in favor of lower latency. The next section summarizes these options.

Data Transfer Assumptions

When processing data transfers between tasks, the code generator makes these assumptions:

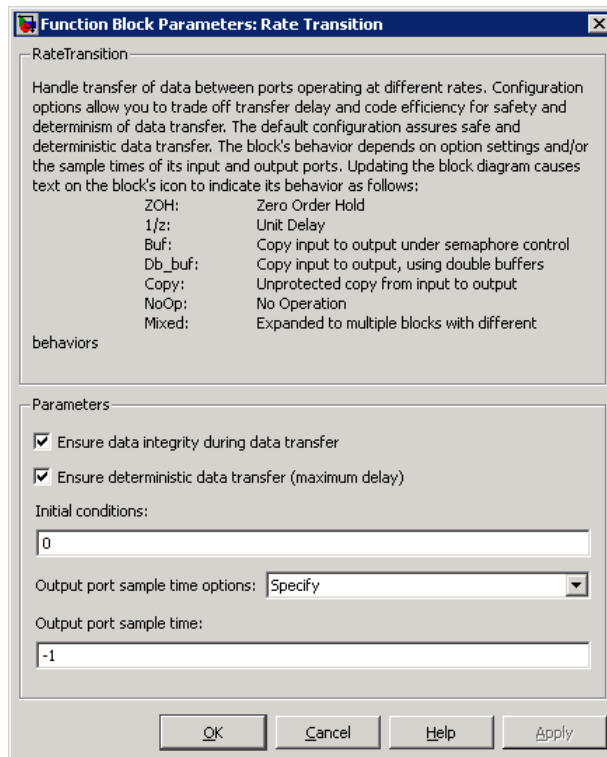
- Data transitions occur between a single reading task and a single writing task.
- A read or write of a byte-sized variable is atomic.
- When two tasks interact through a data transition, only one of them can preempt the other.
- For periodic tasks, the faster rate task has higher priority than the slower rate task; the faster rate task preempts the slower rate task.
- All tasks run on a single processor. Time slicing is not allowed.
- Processes do not crash or restart (especially while data is transferred between tasks).

Rate Transition Block Options

Several parameters of the Rate Transition block are relevant to its use in code generation for real-time execution, as discussed below. For a complete block description, see `Rate Transition` in the Simulink documentation.

The Rate Transition block handles periodic (fast to slow and slow to fast) and asynchronous transitions. When inserted between two blocks of differing sample rates, the Rate Transition block automatically configures its input and output sample rates for the type of transition; you do not need to specify whether a transition is slow-to-fast or fast-to-slow (low-to-high or high-to-low priorities for asynchronous tasks).

The critical decision you must make in configuring a Rate Transition block is the choice of data transfer mechanism to be used between the two rates. Your choice is dictated by considerations of safety, memory usage, and performance. As the Rate Transition block parameter dialog box in the next figure shows, the data transfer mechanism is controlled by two options.



- **Ensure data integrity during data transfer:** When this option is on, data transferred between rates maintains its integrity (the data transfer is protected). When this option is off, the data might not maintain its integrity (the data transfer is unprotected). By default, **Ensure data integrity during data transfer** is on.

- **Ensure deterministic data transfer (maximum delay):** This option is supported for periodic tasks with an offset of zero and fast and slow rates that are multiples of each other. Enable this option for protected data transfers (when **Ensure data integrity during data transfer** is on). When this option is on, the Rate Transition block behaves like a Zero-Order Hold block (for fast to slow transitions) or a Unit Delay block (for slow to fast transitions). The Rate Transition block controls the timing of data transfer in a completely predictable way. When this option is off, the data transfer is nondeterministic. By default, **Ensure deterministic data transfer (maximum delay)** is on for transitions between periodic rates with an offset of zero; for asynchronous transitions, it cannot be selected.

Thus the Rate Transition block offers three modes of operation with respect to data transfer. In order of level of safety:

- **Protected/Deterministic (default):** This is the safest mode. The drawback of this mode is that it introduces deterministic latency into the system for the case of slow-to-fast periodic rate transitions. For that case, the latency introduced by the Rate Transition block is one sample period of the slower task. For the case of fast-to-slow periodic rate transitions, the Rate Transition block introduces no additional latency.
- **Protected/NonDeterministic:** In this mode, for slow-to-fast periodic rate transitions, data integrity is protected by double-buffering data transferred between rates. For fast-to-slow periodic rate transitions, a semaphore flag is used. The blocks downstream from the Rate Transition block use the latest available data from the block that drives the Rate Transition block. Maximum latency is less than or equal to one sample period of the faster task.

The drawbacks of this mode are its nondeterministic timing. The advantage of this mode is its low latency.

- **Unprotected/NonDeterministic:** This mode is not recommended for mission-critical applications. The latency of this mode is the same as for Protected/NonDeterministic mode, but memory requirements are reduced since neither double-buffering nor semaphores are required. That is, the Rate Transition block does nothing in this mode other than to pass signals through; it simply exists to notify you that a rate transition exists (and can cause generated code to compute incorrect answers). Selecting this mode, however, generates the least amount of code.

Note In unprotected mode (**Ensure data integrity during data transfer** option off), the Rate Transition block does nothing other than allow the rate transition to exist in the model.

Rate Transition Blocks and Continuous Time

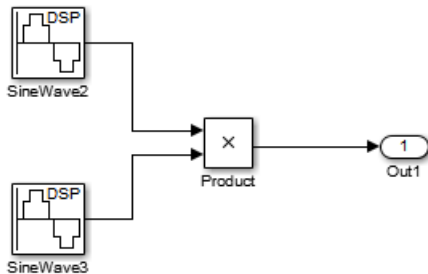
The sample time at the output port of a Rate Transition block can only be discrete or fixed in minor time step. This means that when a Rate Transition block inherits continuous sample time from its destination block, it treats the inherited sample time as Fixed in Minor Time Step. Therefore, the output function of the Rate Transition block runs only at major time steps. If the destination block sample time is continuous, Rate Transition block output sample time is the base rate sample time (if solver is fixed-step), or zero-order-hold-continuous sample time (if solver is variable-step).

Automatic Rate Transition

The Simulink engine can detect mismatched rate transitions in a multitasking model during an update diagram and automatically insert Rate Transition blocks to handle them. To enable this, in the **Solver** pane of model configuration parameters, select **Automatically handle rate transition for data transfer**. The default setting for this option is off. When you select this option:

- Simulink handles transitions between periodic sample times and asynchronous tasks.
- Simulink inserts hidden Rate Transition blocks in the block diagram.
- The code generator produces code for the Rate Transition blocks that were automatically inserted. This code is identical to the code generated for Rate Transition blocks that were inserted manually.
- Automatically inserted Rate Transition blocks operate in protected mode for periodic tasks and asynchronous tasks. You cannot alter this behavior. For periodic tasks, automatically inserted Rate Transition blocks operate with the level of determinism specified by the **Deterministic data transfer** parameter in the **Solver** pane. The default setting is **Whenever possible**, which enables determinism for data transfers between periodic sample-times that are related by an integer multiple. For more information, see “Deterministic data transfer”. To use other modes, you must insert Rate Transition blocks and set their modes manually.

For example, in this model, SineWave2 has a sample time of 2, and SineWave3 has a sample time of 3.

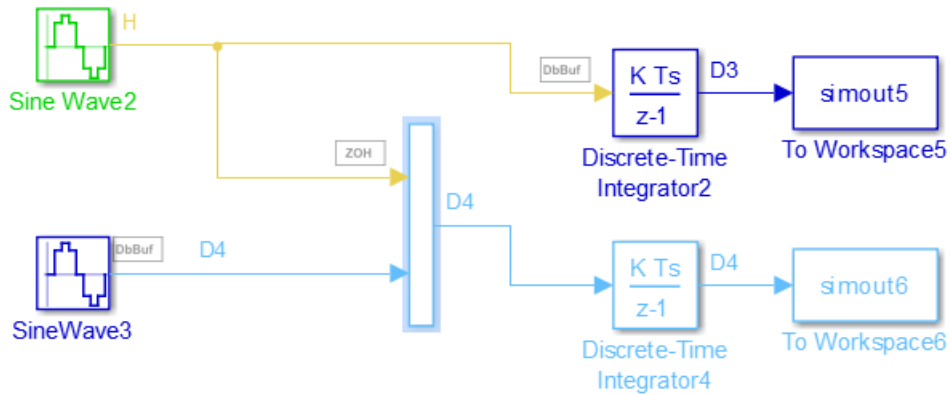


When you select **Automatically handle rate transition for data transfer**, Simulink inserts a Rate Transition block between each Sine Wave block and the Product block. The inserted blocks have the parameter values needed to reconcile the Sine Wave block sample times.

If the input port and output port data sample rates in a model are not multiples of each other, Simulink inserts a Rate Transition block whose sample rate is the greatest common divisor (GCD) of the two rates. If no other block in the model contains this new rate, an error occurs during simulation. In this case, you must insert a Rate Transition block manually.

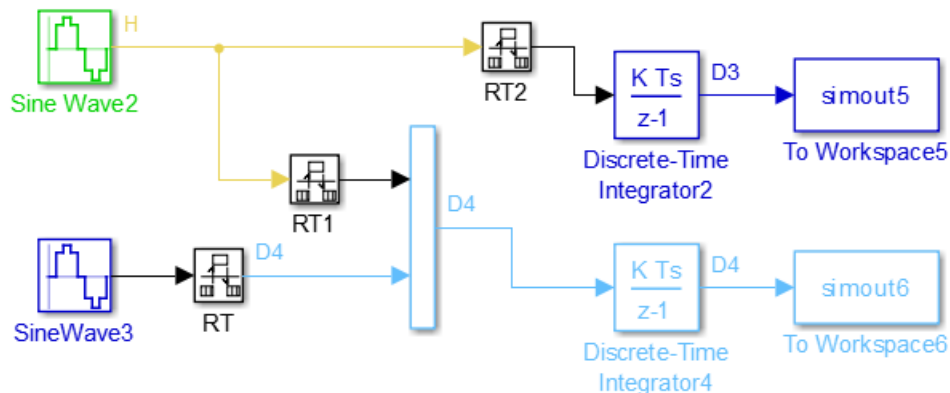
Visualize Inserted Rate Transition Blocks

When you select the **Automatically handle rate transition for data transfer** option, Simulink inserts Rate Transition blocks in the paths that have mismatched transition rates. These blocks are hidden by default. To visualize the inserted blocks, update the diagram. Badge labels appear in the model and indicate where Simulink inserted Rate Transition blocks during the compilation phase. For example, in this model, three Rate Transition blocks were inserted between the two Sine Wave blocks and the Multiplexer and Integrator when the model compiled. The ZOH and DbBuf badge labels indicate these blocks.



You can show or hide badge labels using the **Display > Signals and Ports > Hidden Rate Transition Block Indicators** setting.

To configure the hidden Rate Transition blocks, right click on a badge label and click on **Insert rate transition block** to make the block visible.

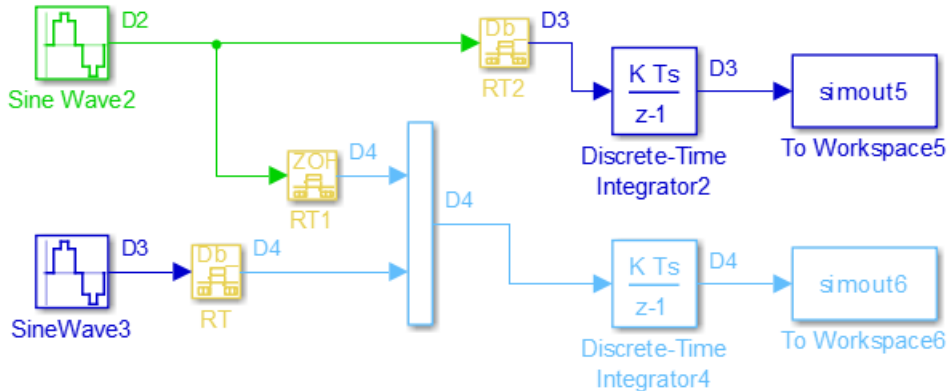


When you make hidden Rate Transition blocks visible:

- You can see the type of Rate Transition block inserted as well as the location in the model.
- You can set the **Initial Conditions** of these blocks.

- You can change block parameters for rate transfer.

Validate the changes to your model by updating your diagram.



Displaying inserted Rate Transition blocks is not compatible with:

- Concurrent execution environment
- Export-function models

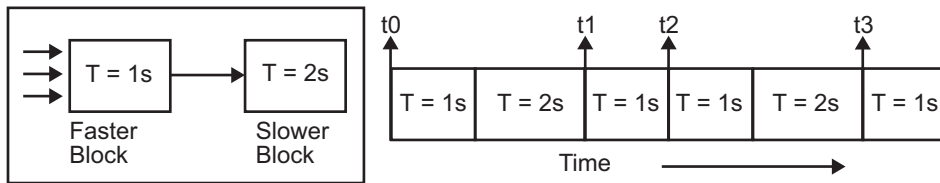
To learn more about the types of Rate Transition blocks, see [Rate Transition](#).

Periodic Sample Rate Transitions

These sections describe cases in which Rate Transition blocks are required for periodic sample rate transitions. The discussion and timing diagrams in these sections are based on the assumption that the Rate Transition block is used in its default (protected/deterministic) mode; that is, the **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** options are both on. These are the settings used for automatically inserted Rate Transition blocks.

Faster to Slower Transitions in a Simulink Model

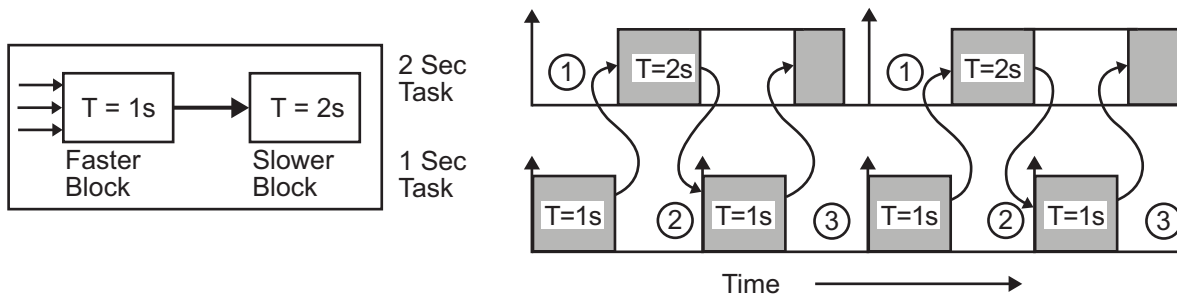
In a model where a faster block drives a slower block having direct feedthrough, the outputs of the faster block are computed first. In simulation intervals where the slower block does not execute, the simulation progresses more rapidly because there are fewer blocks to execute. The next figure illustrates this situation.



A Simulink simulation does not execute in real time, which means that it is not bound by real-time constraints. The simulation waits for, or moves ahead to, whatever tasks are required to complete simulation flow. The actual time interval between sample time steps can vary.

Faster to Slower Transitions in Real Time

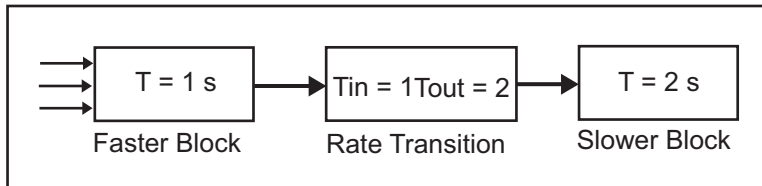
In models where a faster block drives a slower block, you must compensate for the fact that execution of the slower block might span more than one execution period of the faster block. This means that the outputs of the faster block can change before the slower block has finished computing its outputs. The next figure shows a situation in which this problem arises ($T =$ sample time). Note that lower priority tasks are preempted by higher priority tasks before completion.



- ① The faster task ($T=1s$) completes.
- ② Higher priority preemption occurs.
- ③ The slower task ($T=2s$) resumes and its inputs have changed. This leads to unpredictable results.

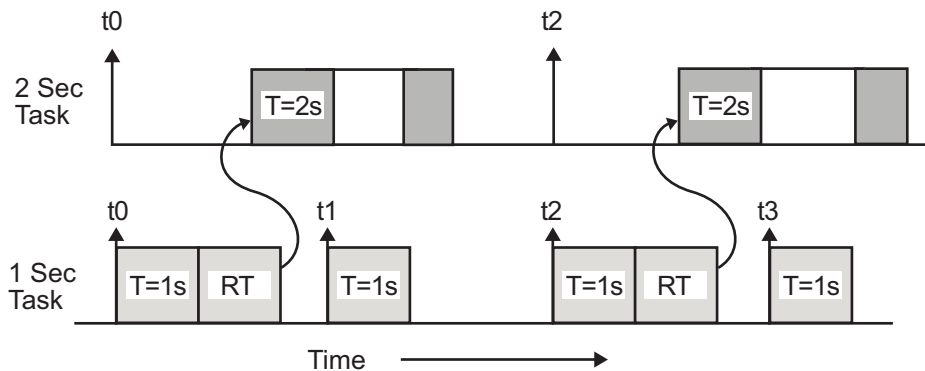
In the above figure, the faster block executes a second time before the slower block has completed execution. This can cause unpredictable results because the input data to the slow task is changing. Data might not maintain its integrity in this situation.

To avoid this situation, the Simulink engine must hold the outputs of the 1 second (faster) block until the 2 second (slower) block finishes executing. The way to accomplish this is by inserting a Rate Transition block between the 1 second and 2 second blocks. The input to the slower block does not change during its execution, maintaining data integrity.



It is assumed that the Rate Transition block is used in its default (protected/ deterministic) mode.

The Rate Transition block executes at the sample rate of the slower block, but with the priority of the faster block.

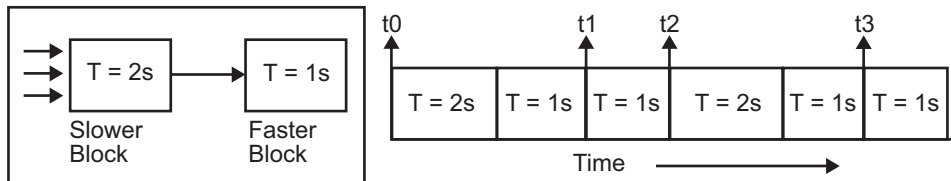


When you add a Rate Transition block, the block executes before the 2 second block (its priority is higher) and its output value is held constant while the 2 second block executes (it executes at the slower sample rate).

Slower to Faster Transitions in a Simulink Model

In a model where a slower block drives a faster block, the Simulink engine again computes the output of the driving block first. During sample intervals where only the faster block executes, the simulation progresses more rapidly.

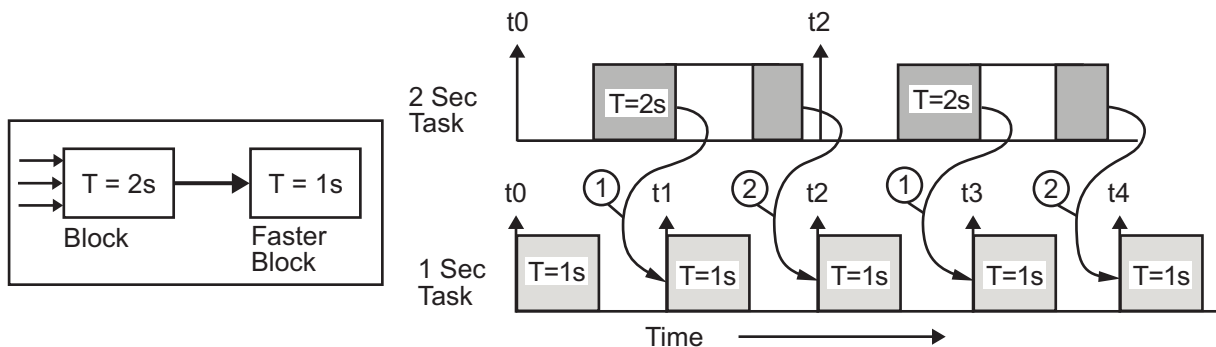
The next figure shows the execution sequence.



As you can see from the preceding figures, the Simulink engine can simulate models with multiple sample rates in an efficient manner. However, a Simulink simulation does not operate in real time.

Slower to Faster Transitions in Real Time

In models where a slower block drives a faster block, the generated code assigns the faster block a higher priority than the slower block. This means the faster block is executed before the slower block, which requires special care to avoid incorrect results.



- ① The faster block executes a second time prior to the completion of the slower block.
- ② The faster block executes before the slower block.

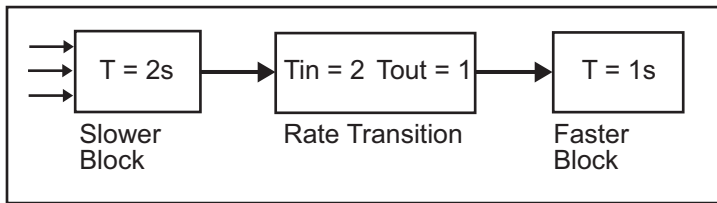
This timing diagram illustrates two problems:

- Execution of the slower block is split over more than one faster block interval. In this case the faster task executes a second time before the slower task has completed

execution. This means the inputs to the faster task can have incorrect values some of the time.

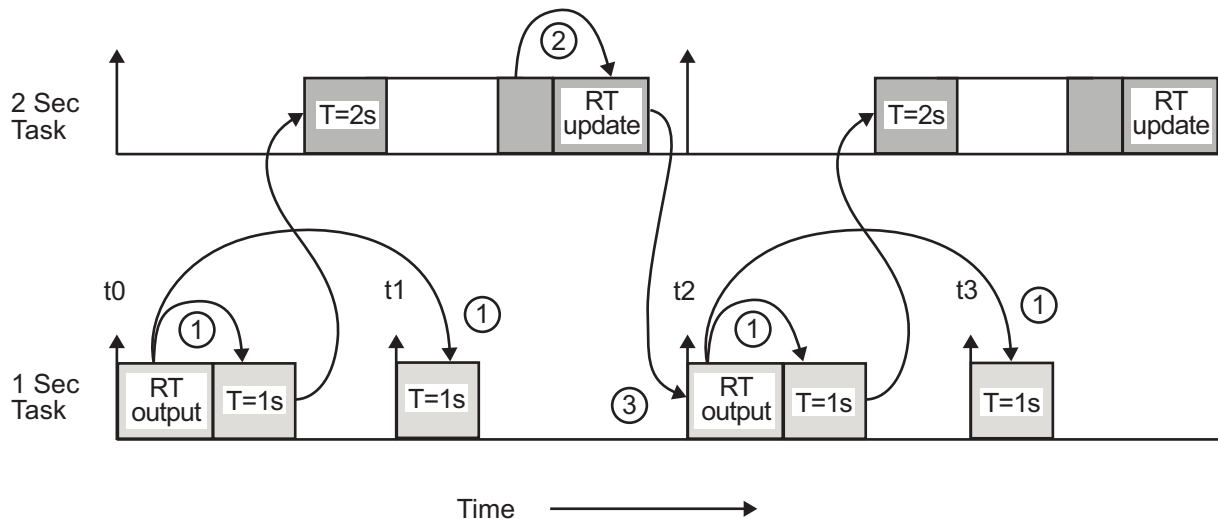
- The faster block executes before the slower block (which is backward from the way a Simulink simulation operates). In this case, the 1 second block executes first; but the inputs to the faster task have not been computed. This can cause unpredictable results.

To eliminate these problems, you must insert a Rate Transition block between the slower and faster blocks.



It is assumed that the Rate Transition block is used in its default (protected/deterministic) mode.

The next figure shows the timing sequence that results with the added Rate Transition block.



Three key points about transitions in this diagram (refer to circled numbers):

- 1 The Rate Transition block output runs in the 1 second task, but at a slower rate (2 seconds). The output of the Rate Transition block feeds the 1 second task blocks.
- 2 The Rate Transition update uses the output of the 2 second task to update its internal state.
- 3 The Rate Transition output in the 1 second task uses the state of the Rate Transition that was updated in the 2 second task.

The first problem is alleviated because the Rate Transition block is updating at a slower rate and at the priority of the slower block. The input to the Rate Transition block (which is the output of the slower block) is read after the slower block completes executing.

The second problem is alleviated because the Rate Transition block executes at a slower rate and its output does not change during the computation of the faster block it is driving. The output portion of a Rate Transition block is executed at the sample rate of the slower block, but with the priority of the faster block. Since the Rate Transition block drives the faster block and has effectively the same priority, it is executed before the faster block.

Note This use of the Rate Transition block changes the model. The output of the slower block is now delayed by one time step compared to the output without a Rate Transition block.

More About

- “Time-Based Scheduling and Code Generation” on page 4-2
- “Sample Times in Subsystems”
- “Sample Times in Systems”
- “Modeling for Multitasking Execution” on page 4-12
- “Configure Time-Based Scheduling” on page 4-34
- “Resolve Rate Transitions”
- “Time-Based Scheduling Example Models” on page 4-36

Configure Time-Based Scheduling

For details about solver options, see “Solver Pane” in the Simulink reference documentation.

Configure Start and Stop Times

The **Stop time** must be greater than or equal to the **Start time**. If the stop time is zero, or if the total simulation time (Stop minus Start) is less than zero, the generated program runs for one step. If the stop time is set to `inf`, the generated program runs indefinitely.

When using the GRT or ERT targets, you can override the stop time when running a generated program from the Microsoft Windows command prompt or UNIX¹ command line. To override the stop time that was set during code generation, use the `-tf` switch.

```
model -tf n
```

The program runs for `n` seconds. If `n = inf`, the program runs indefinitely.

Certain blocks have a dependency on absolute time. If you are designing a program that is intended to run indefinitely (**Stop time** = `inf`), and your generated code does not use the `rtModel` data structure (that is, it uses `simstructs` instead), you must not use these blocks. See “Absolute Time Limitations” on page 3-12 for a list of blocks that can potentially overflow timers.

If you know how long an application that depends on absolute time needs to run, you can prevent the timers from overflowing and force the use of optimal word sizes by specifying the **Application lifespan (days)** parameter on the **Optimization** pane. See “Control Memory Allocation for Time Counters” on page 31-15 for details.

Configure the Solver Type

For code generation, you must configure a model to use a fixed-step solver for all targets except the S-function and RSim targets. You can configure the S-function and RSim targets with a fixed-step or variable-step solver.

1. UNIX is a registered trademark of The Open Group in the United States and other countries.

Configure the Tasking Mode

The code generator supports both single-tasking and multitasking modes for periodic sample times. See “Time-Based Scheduling and Code Generation” on page 4-2 for details.

More About

- “Time-Based Scheduling and Code Generation” on page 4-2
- “Sample Times in Subsystems”
- “Sample Times in Systems”
- “Time-Based Scheduling Example Models” on page 4-36

Time-Based Scheduling Example Models

Optimize Memory Usage for Time Counters

This example shows how to optimize the amount of memory that the code generator allocates for time counters. The example optimizes the memory that stores elapsed time, the interval of time between two events.

The code generator represents time counters as unsigned integers. The word size of time counters is based on the setting of the model configuration parameter **Application lifespan (days)**, which specifies the expected maximum duration of time the application runs. You can use this parameter to prevent time counter overflows. The default size is 64 bits.

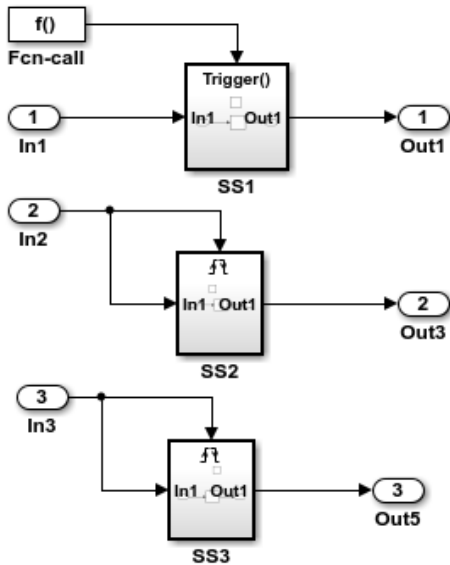
The number of bits that a time counter uses depends on the setting of the **Application lifespan (days)** parameter. For example, if a time counter increments at a rate of 1 kHz, to avoid an overflow, the counter has the following number of bits:

- Lifespan < 0.25 sec: 8 bits
- Lifespan < 1 min: 16 bits
- Lifespan < 49 days: 32 bits
- Lifespan > 50 days: 64 bits

A 64-bit time counter does not overflow for 590 million years.

Open Example Model

Open the example model `rtwdemo_abstime`.



SS1 is clocked at 1 kHz, and contains a discrete-time integrator that requires elapsed time to compute its output. However, a counter is not required to compute elapsed time since the trigger port 'Sample time type' is set to 'periodic.' Instead, time is inlined as 1 kHz.

SS2 is clocked at 100 Hz, and contains a discrete-time integrator that requires elapsed time to compute its output. Since the application life span of the model is 1 day, a 32-bit counter is required to compute elapsed time for SS2.

SS3 is clocked at 0.5 Hz, and contains a discrete-time integrator that requires elapsed time to compute its output. Since the application life span of the model is 1 day, a 16-bit counter is required to compute elapsed time for SS3.

Did you know ...

Display Sample Time Colors (double-click)

Simulink Coder optimizes how counters are employed to measure absolute and elapsed time:

- o Time is computed from unsigned integer counters.
- o Only tasks that require time are allocated a counter.
- o Elapsed time is computed by a subsystem if and only if a block in its hierarchy requires elapsed time.
- o Time is shared by all blocks within a triggered hierarchy.

Simulink Coder further optimizes counters based on the option "Application life span," whereby the number of bits used for a particular counter is optimized based on how long the application will run.

Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

Copyright 1994-2012 The MathWorks, Inc.

The model consists of three subsystems SS1, SS2, and SS3. On the **Optimization** tab, the **Application lifespan (days)** parameter is set to the default, which is **auto**.

The three subsystems contain a discrete-time integrator that requires elapsed time as input to compute its output value. The subsystems vary as follows:

- SS1 - Clocked at 1 kHz. Does not require a time counter. **Sample time type** parameter for trigger port is set to `periodic`. Elapsed time is inlined as 0.001.
- SS2 - Clocked at 100 Hz. Requires a time counter. Based on a lifespan of 1 day, a 32-bit counter stores the elapsed time.
- SS3 - Clocked at 0.5 Hz. Requires a time counter. Based on a lifespan of 1 day, a 16-bit counter stores the elapsed time.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for the three subsystems appear red, green, and blue. Triggered subsystems are blue-green.

Generate Code and Report

1. Create a temporary folder for the build and inspection process.
2. Configure the model for the code generator to use the GRT system target file and a lifespan of `inf` days.
3. Build the model.

```
### Starting build procedure for model: rtwdemo_abstime  
### Successful completion of build procedure for model: rtwdemo_abstime
```

Review Generated Code

Open the generated source file `rtwdemo_abstime.c`.

```
struct tag_RTM_rtwdemo_abstime_T {  
    const char_T *errorStatus;  
  
    /*  
     * Timing:  
     * The following substructure contains information regarding  
     * the timing information for the model.  
     */  
    struct {  
        uint32_T clockTick1;  
        uint32_T clockTickH1;  
        uint32_T clockTick2;  
        uint32_T clockTickH2;
```

```

    struct {
        uint16_T TID[3];
        uint16_T cLimit[3];
    } TaskCounters;
} Timing;
};

```

Four 32-bit unsigned integers, `clockTick1`, `clockTickH1`, `clockTick2`, and `clockTickH2` are counters for storing the elapsed time of subsystems SS2 and SS3.

Enable Optimization and Regenerate Code

1. Reconfigure the model to set the lifespan to 1 day.
2. Build the model.

```

### Starting build procedure for model: rtwdemo_abstime
### Successful completion of build procedure for model: rtwdemo_abstime

```

Review the Regenerated Code

```

struct tag_RTM_rtwdemo_abstime_T {
    const char_T *errorStatus;

    /*
     * Timing:
     * The following substructure contains information regarding
     * the timing information for the model.
     */
    struct {
        uint32_T clockTick1;
        uint16_T clockTick2;
        struct {
            uint16_T TID[3];
            uint16_T cLimit[3];
        } TaskCounters;
    } Timing;
};

```

The new setting for the **Application lifespan (days)** parameter instructs the code generator to set aside less memory for the time counters. The regenerated code includes:

- 32-bit unsigned integer, `clockTick1`, for storing the elapsed time of the task for SS2
- 16-bit unsigned integer, `clockTick2`, for storing the elapsed time of the task for SS3

Related Information

- “Optimization Pane: General”
- “Timers in Asynchronous Tasks”
- “Time-Based Scheduling and Code Generation”

Single-Rate Modeling (Bare Board, No OS)

This model shows the code generated for a single-rate discrete-time model configured for a bare-board target (one with no operating system).

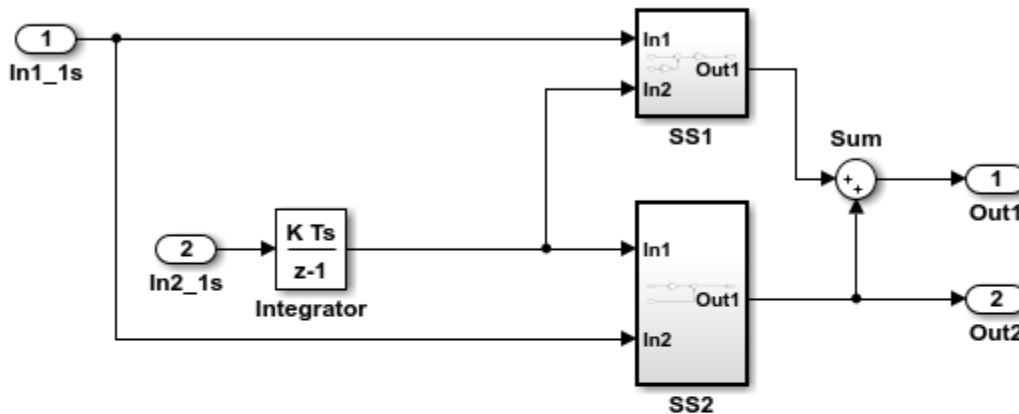
Open Example Model

Open the example model `rtwdemo_srbb`.

```
open_system('rtwdemo_srbb')
```

This model is configured to display sample-time colors upon diagram update. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times. Click the yellow button to the left to update the diagram and show sample-time colors.

Display Sample
Time Colors
(double-click)



This model shows the code generated for a single-rate discrete-time model configured for a bare-board target (one with no operating system). The model uses one sample time. Inport block 1 and Inport block 2 both specify a 1-second sample time, which is enforced by the "Periodic sample time constraint" option on the Solver configuration page. To view the solver page, double-click the yellow button below. To display the sample times in the model, double-click the yellow button above.

Generate Code Using
Simulink Coder
(double-click)

Generate Code Using
Embedded Coder
(double-click)

View Solver
Configuration
(double-click)

Copyright 1994-2012 The MathWorks, Inc.

The model uses one sample time. Inport block 1 and Inport block 2 both specify a 1-second sample time, which is enforced by the **Periodic sample time constraint** option

on the **Solver** configuration page. To view the solver page, double-click the corresponding yellow button in the model. To display the sample times in the model, double-click the corresponding yellow button in the model.

This model is configured to display sample-time colors upon diagram update. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times. Click the yellow button in the model to update the diagram and show sample-time colors.

Multirate Modeling in Single-Tasking Mode (Bare Board, no OS)

This model shows the code generated for a multirate discrete-time model configured for single-tasking on a bare-board target (one with no operating system).

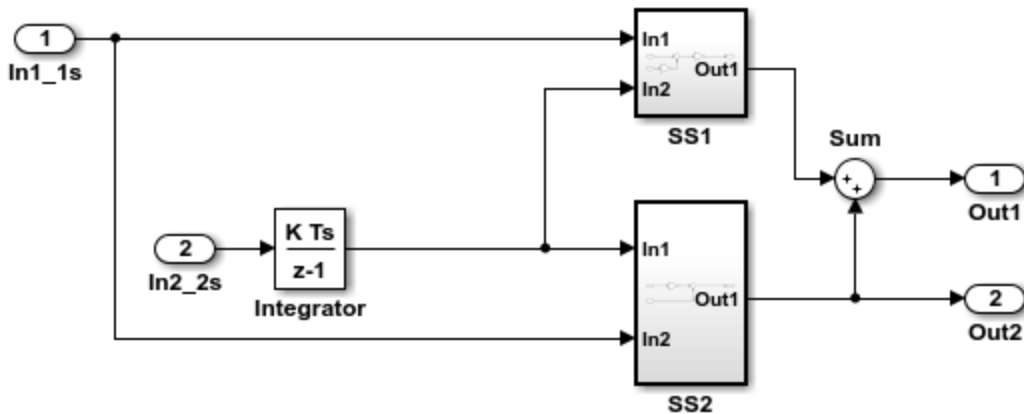
Open Example Model

Open the example model `rtwdemo_mrstbb`.

```
open_system('rtwdemo_mrstbb')
```


The model is configured to display sample-time colors upon diagram update. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times. Click the yellow button to the right to update the diagram and show sample-time colors.

**Display Sample
Time Colors
(double-click)**



This model shows the code generated for a multirate discrete-time model configured for single-tasking on a bare-board target (one with no operating system). The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the "Periodic sample time constraint" option on the Solver configuration page. The solver is set for single-tasking operation. Rate transition blocks are, therefore, not necessary between blocks executing at different sample times because preemption will not occur.

**Generate Code Using
Simulink Coder
(double-click)**

**Generate Code Using
Embedded Coder
(double-click)**

**View Solver
Configuration
(double-click)**

The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the **Periodic sample time constraint** option on the **Solver** configuration page. The solver is set for single-tasking operation. Rate transition blocks are, therefore, not necessary between blocks executing at different sample times because preemption will not occur.

The model is configured to display sample-time colors upon diagram update. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times. Double-click the yellow button in the model to update the diagram and show sample-time colors.

Multirate Modeling in Multitasking Mode (Bare Board, no OS)

This model shows the code generated for a multirate discrete-time model configured for a multitasking bare-board target (one with no operating system).

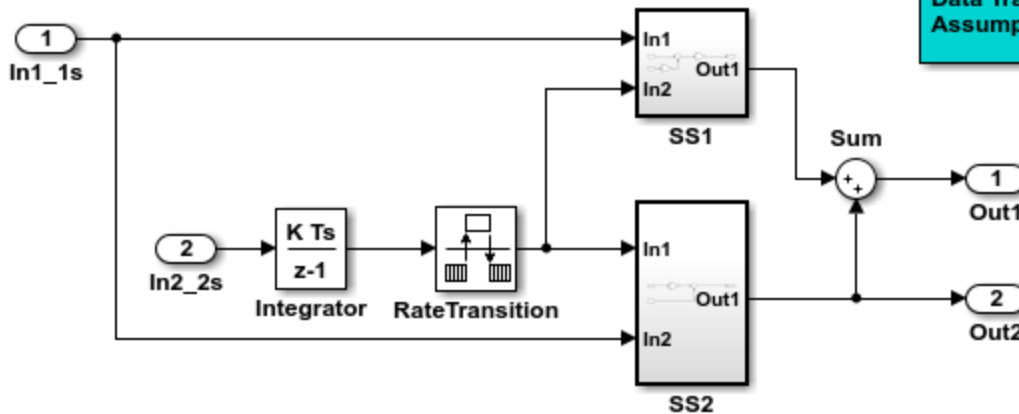
Open Example Model

Open the example model `rtwdemo_mrmtbb`.

```
open_system('rtwdemo_mrmtbb')
```

The model is configured to display sample-time colors upon diagram update. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times. Click the yellow button to the right to update the diagram and show sample-time colors.

Display Sample
Time Colors
(double-click)



Data Transfer
Assumptions ...

This model shows the code generated for a multirate discrete-time model configured for a multitasking bare-board target (one with no operating system). The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the "Periodic sample time constraint" option on the Solver configuration page. The solver is set for multitasking operation, which means a rate transition block is required to ensure that data integrity is enforced when the 1-second task preempts the 2-second task. Proper rate transitions are always enforced by Simulink and Simulink Coder. This model specifies an explicit rate transition block. Alternatively, this block could be automatically inserted by Simulink using the "Automatically handle data transfers between tasks" option on the Solver configuration page.

Generate Code Using
Simulink Coder
(double-click)

Generate Code Using
Embedded Coder
(double-click)

View Solver
Configuration
(double-click)

Explore Example Model

The model contains two sample times. Inport block 1 and Inport block 2 specify 1-second and 2-second sample times, respectively, which are enforced by the **Periodic sample time constraint** option on the **Solver** configuration page. The solver is set for multitasking operation, which means a rate transition block is required to ensure that data integrity is enforced when the 1-second task preempts the 2-second task. Proper rate transitions are always enforced by Simulink and Simulink Coder. This model specifies an explicit rate transition block. Alternatively, this block could be automatically inserted by Simulink using the **Automatically handle data transfers between tasks** option on the **Solver** configuration page.

The model is configured to display sample-time colors upon diagram update. Red represents the fastest discrete sample time in the model, green represents the second fastest, and yellow represents mixed sample times. Click the yellow button to the right to update the diagram and show sample-time colors.

Data Transfer Assumptions

Basis of operation for data transfers between tasks:

- 1 Data transitions occur between a single reading task and a single writing task.
- 2 A read or write of a byte sized variable is atomic.
- 3 When two tasks interact through a data transition, only one of them can preempt the other.
- 4 For periodic tasks, the faster rate task has higher priority than the slower rate task; the faster rate task always preempts the slower rate task.
- 5 All tasks run on a single processor. Time slicing is not allowed.
- 6 Processes do not crash/restart (especially while data is being transferred between tasks)

Trade Determinism and Data Integrity to Improve System Performance

This model shows the differences in the operation modes of the Rate Transition block when used in a multirate, multitasking model. The flexible options for the Rate Transition block allow you to select the mode that is best suited for your application. You can trade levels of determinism and data integrity to improve system performance.

Rate Transition Block Modes of Operation

Ensure data integrity and determinism (DetAndInteg) : Data is transferred such that all data bytes for the signal (including all elements of a wide signal) are from the same time step. Additionally, it is ensured that the relative sample time (delay) from which the data is transferred from one rate to another is always the same. Only ANSI-C code is used, no target specific 'critical section' protection is needed.

Ensure integrity (IntegOnly) : Data is transferred such that all data bytes for the signal (including all elements of a wide signal) are from the same time step. However, from one transfer of data to the next, the relative sample time (delay) for which the data is transferred can vary. In this mode, the code to read/write the data is run more often than in the DetandInt mode. In the worst case, the delay is equivalent to the DetandInt mode, but the delay can be less which is important in some applications. Also, this mode supports data transfers to/from asynchronous rates which the DetandInt mode cannot support. Only ANSI-C code is used, no target specific 'critical section' protection is needed.

No data consistency operations are performed (None) : For this case, the Rate Transition block does not generate code. This mode is acceptable in some applications where atomic access of scalar data types is guaranteed and when the relative temporal values of the data is not important. This mode does not introduce any delay.

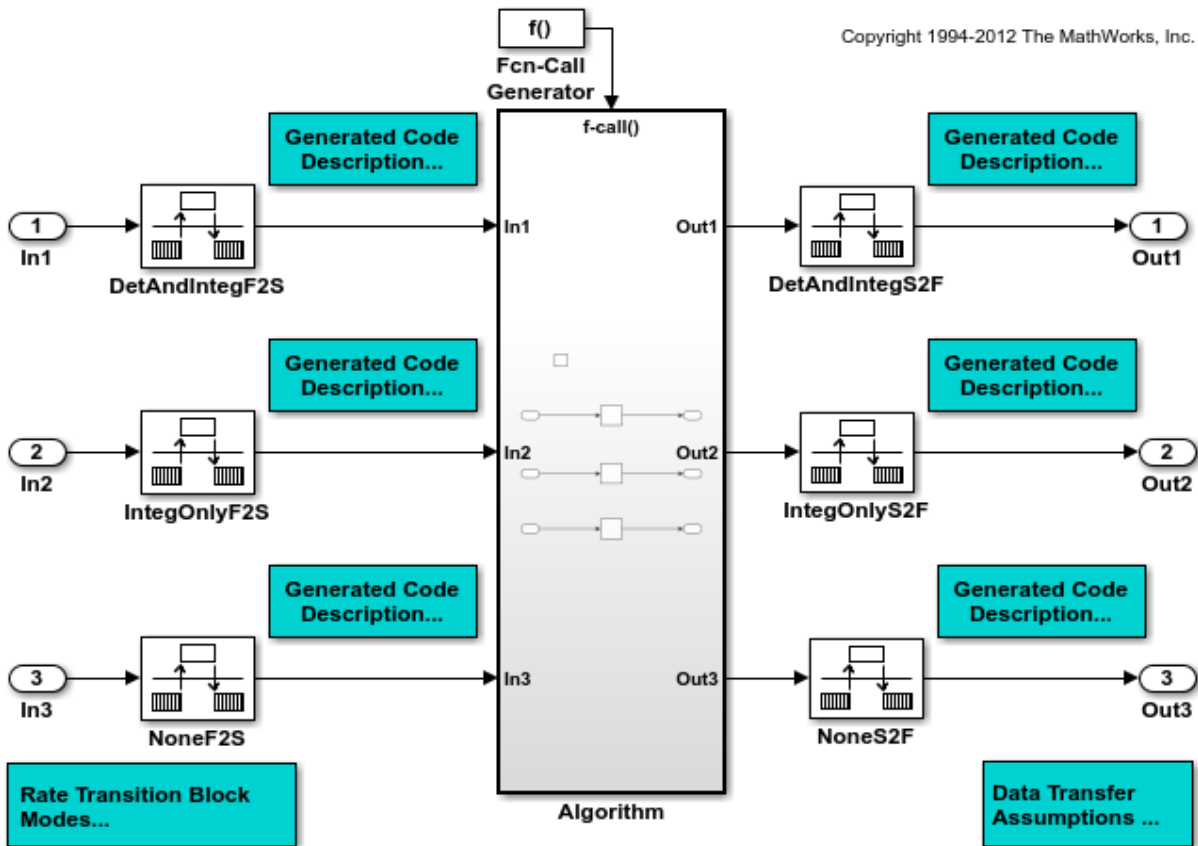
Data Transfer Assumptions

Basis of operation for data transfers between tasks:

- Data transitions occur between a single reading task and a single writing task.
- A read or write of a byte sized variable is atomic.
- When two tasks interact through a data transition, only one of them can preempt the other.
- For periodic tasks, the faster rate task has higher priority than the slower rate task; the faster rate task always preempts the slower rate task.
- All tasks run on a single processor. Time slicing is not allowed.
- Processes do not crash/restart (especially while data is being transferred between tasks)

Model `rtwdemo_ratetrans`

```
open_system('rtwdemo_ratetrans')
```



This model shows the differences in the operation modes of the Rate Transition block when used in a multirate, multitasking model. The flexible options for the Rate Transition block allow you to select the mode that is best suited for your application. You can trade levels of determinism and data integrity to improve system performance.

Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

Display Sample Time Colors (double-click)

Model `rtwdemo_ratetrans` shows the differences in the operation modes of the following Rate Transition blocks.

Rate Transition block `DetAndIntegF2S`

Determinism and data integrity (fast to slow transition):

- The block output is used as a persistent data buffer.
- Data is written to output at slower rate but done during the faster rate context
- Data as seen by the slower rate is always the value when both the faster and slower rate last executed. Any subsequent steps by the faster rate (and associated data updates) while the slower rate is running are not seen by the slower rate.

Rate Transition block `DetAndIntegS2F`

Determinism and data integrity (slow to fast transition):

- Uses two persistent data buffers, an internal buffer and the blocks output.
- The internal buffer is copied to the output at the slower rate but done during the faster rate context.
- The internal buffer is written at the slower rate and during the slower rate context.
- The data that Fast rate sees is always delayed, i.e. data is from the previous step of the slow rate code.

Rate Transition block `IntegOnlyF2S`

Data integrity only (fast to slow transition):

- The block output is used as a persistent data buffer.
- Data is written to buffer during the faster rate context if a flag indicates it not in the process of being read.
- The flag is set and data is copied from the buffer to output at the slow rate, the flag is then cleared. This is an additional copy as compared to the deterministic case.
- Data as seen by the slower rate can be from a more recent step of the faster rate than from when the slower rate and faster rate both executed.

Rate Transition block `IntegOnlyS2F`

Data integrity only (slow to fast transition):

- Uses two persistent data buffers, both are internal buffers.

- One of the 2 buffers is always copied to the output at faster rate.
- One of the 2 buffers is written at the slower rate and during the slower rate context, then the active buffer is switched.
- The data as seen by the faster rate can be more recent than for the deterministic case. Specifically, when both the slower and faster rate have their hits, the faster rate will see a previous value from the slower rate. But, subsequent steps for the faster rate may see an updated value (when the slower rate updates the non-active buffer and switches the active buffer flag).

Rate Transition block NoneF2S

No code is generated for the Rate Transition block when determinism and data integrity is waived.

Rate Transition block NoneS2F

No code is generated for the Rate Transition block when determinism and data integrity is waived.

```
bdclose('rtwdemo_ratetrans');
```

More About

- “Time-Based Scheduling and Code Generation” on page 4-2
- “Modeling for Single-Tasking Execution” on page 4-8
- “Modeling for Multitasking Execution” on page 4-12

Event-Based Scheduling in Simulink Coder

- “Asynchronous Events” on page 5-2
- “Generate Interrupt Service Routines” on page 5-6
- “Spawn and Synchronize Execution of RTOS Task” on page 5-14
- “Pass Asynchronous Events in RTOS as Input To a Referenced Model” on page 5-30
- “Rate Transitions and Asynchronous Blocks” on page 5-37
- “Timers in Asynchronous Tasks” on page 5-42
- “Create a Customized Asynchronous Library” on page 5-45
- “Import Asynchronous Event Data for Simulation” on page 5-53
- “Asynchronous Support Limitations” on page 5-57

Asynchronous Events

Asynchronous Support

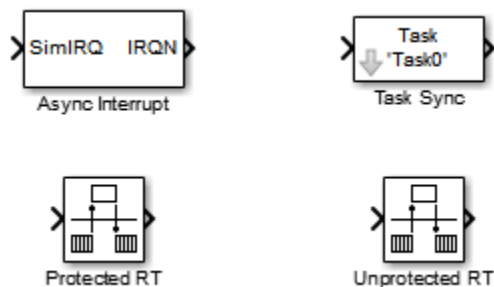
Normally, you time models from which you plan to generate code from a *periodic* interrupt source (for example, a hardware timer). Blocks in a periodically clocked single-rate model run at a timer interrupt rate (the base rate of the model). Blocks in a periodically clocked multirate model run at the base rate or at multiples of that rate.

Many systems must also support execution of blocks in response to events that are *asynchronous* with respect to the periodic timing source of the system. For example, a peripheral device might signal completion of an input operation by generating an interrupt. The system must service such interrupts, for example, by acquiring data from the interrupting device.

This chapter explains how to use blocks to model and generate code for asynchronous event handling, including servicing of hardware-generated interrupts, maintenance of timers, asynchronous read and write operations, and spawning of asynchronous tasks under a real-time operating system (RTOS). This block library demonstrates integration with an example RTOS (VxWorks[®]). Although the blocks target an example RTOS, this chapter provides source code analysis and other information you can use to develop blocks that support asynchronous event handling for an alternative target RTOS.²

Block Library for Calls to an Example Real-Time Operating System

The next figure shows the blocks in the vxlib1 block library.



2. VxWorks is a registered trademark of Wind River[®] Systems, Inc.

The key blocks in the library are the Async Interrupt and Task Sync blocks. These blocks are targeted for an example RTOS (VxWorks). You can use them, with modification, to support your RTOS applications.

Note: You can use the blocks in the `vxlib1` on page 5-2 library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for your target environment.

To implement asynchronous support for an RTOS other than the example RTOS, use the guidelines and example code are provided to help you adapt the `vxlib1` library blocks to target your RTOS. This topic is discussed in “Create a Customized Asynchronous Library” on page 5-45.

The `vxlib1` library includes blocks you can use to

- Generate interrupt-level code — Async Interrupt block
- Spawn an RTOS task that calls a function call subsystem — Task Sync block
- Enable data integrity when transferring data between blocks running as different tasks — Protected RT block
- Use an unprotected/nondeterministic mode when transferring data between blocks running as different tasks — Unprotected RT block

The use of protected and unprotected Rate Transition blocks in asynchronous contexts is discussed in “Rate Transitions and Asynchronous Blocks” on page 5-37. For general information on rate transitions, see “Time-Based Scheduling and Code Generation” on page 4-2.

Access the Block Library for RTOS Integration

To access the example RTOS (VxWorks) block library, enter the MATLAB command `vxlib1`.

Generate Code Using Library Blocks for RTOS Integration

To generate an example RTOS compatible application from a model containing `vxlib1` library blocks, use the following configuration parameter values for your model.

- Select system target file `ert.tlc` (requires an Embedded Coder license) from the browse menu for the **Code Generation > System target file** parameter (`SystemTargetFile`).
- Enable the **Code Generation > Generate code only** parameter (`GenCodeOnly`).
- Enable the **All Parameters > Generate an example main program** parameter (`GenerateSampleERTMain`).
- Select `VxWorksExample` from the menu for the **All Parameters > Target operating system** parameter (`TargetOS`).

Examples and Additional Information

Additional information relevant to the topics in this chapter can be found in

- The `rtwdemo_async` model, which uses the `tornado.tlc` system target file and `vxlib1` block library. To open this example, type `rtwdemo_async` at the MATLAB command prompt.
- The `rtwdemo_async_mdltreftop` model, which uses the `tornado.tlc` system target file and `vxlib1` block library. To open this example, type `rtwdemo_async_mdltreftop` at the MATLAB command prompt.
- “Time-Based Scheduling and Code Generation” on page 4-2, discusses general multitasking and rate transition issues for periodic models.
- The Embedded Coder documentation discusses the `ert.tlc` system target file, including task execution and scheduling.
- For detailed information about the system calls to the example RTOS (VxWorks) mentioned in this chapter, see VxWorks system documentation on the Wind River website.

More About

- “Time-Based Scheduling and Code Generation” on page 4-2
- “Generate Interrupt Service Routines” on page 5-6
- “Spawn and Synchronize Execution of RTOS Task” on page 5-14
- “Pass Asynchronous Events in RTOS as Input To a Referenced Model” on page 5-30
- “Timers in Asynchronous Tasks” on page 5-42
- “Import Asynchronous Event Data for Simulation” on page 5-53

- “Rate Transitions and Asynchronous Blocks” on page 5-37
- “Create a Customized Asynchronous Library” on page 5-45
- “Asynchronous Support Limitations” on page 5-57

Generate Interrupt Service Routines

To generate an interrupt service routine (ISR) associated with a specific VME interrupt level for the example RTOS (VxWorks), use the **Async Interrupt** block. The Async Interrupt block enables the specified interrupt level and installs an ISR that calls a connected function call subsystem.

You can also use the Async Interrupt block in a simulation. It provides an input port that can be enabled and connected to a simulated interrupt source.

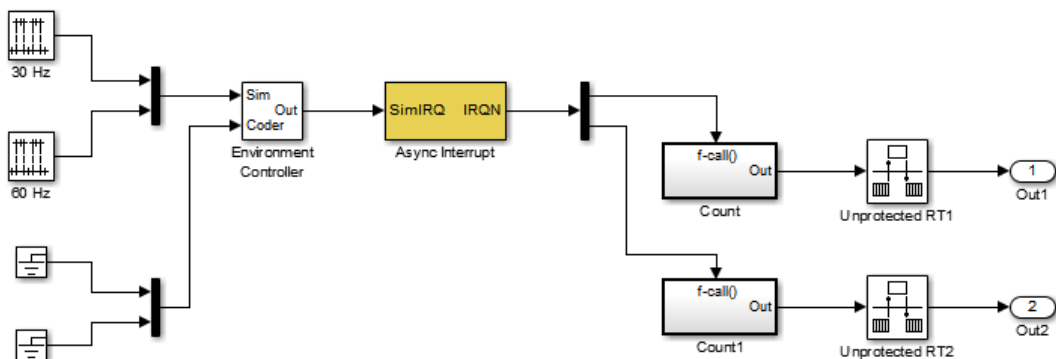
Note: The operating system integration techniques that are demonstrated in this section use one or more blocks the blocks in the vxlib1 on page 5-2 library. These blocks provide starting point examples to help you develop custom blocks for your target environment.

Connecting the Async Interrupt Block

To generate an ISR, connect an output of the Async Interrupt block to the control input of

- A function call subsystem
- The input of a Task Sync block
- The input to a Stateflow chart configured for a function call input event

The next figure shows an Async Interrupt block configured to service two interrupt sources. The outputs (signal width 2) are connected to two function call subsystems.



Requirements and Restrictions

Note the following requirements and restrictions:

- The Async Interrupt block supports VME interrupts 1 through 7.
- The Async Interrupt block uses the following system calls to the example RTOS (VxWorks):
 - `sysIntEnable`
 - `sysIntDisable`
 - `intConnect`
 - `intLock`
 - `intUnlock`
 - `tickGet`

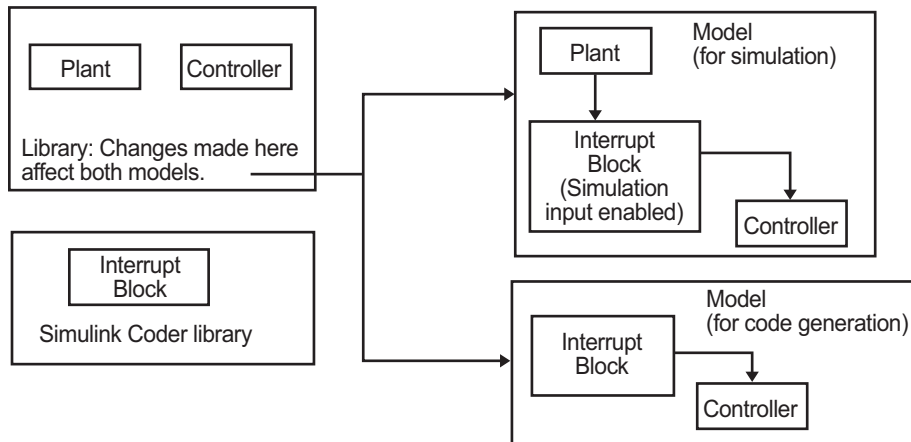
Performance Considerations

Execution of large subsystems at interrupt level can have a significant impact on interrupt response time for interrupts of equal and lower priority in the system. As a general rule, it is best to keep ISRs as short as possible. Connect only function call subsystems that contain a small number of blocks to an Async Interrupt block.

A better solution for large subsystems is to use the Task Sync block to synchronize the execution of the function call subsystem to a RTOS task. The Task Sync block is placed between the Async Interrupt block and the function call subsystem. The Async Interrupt block then installs the Task Sync block as the ISR. The ISR releases a synchronization semaphore (performs a `semGive`) to the task, and returns immediately from interrupt level. The task is then scheduled and run by the example RTOS (VxWorks). See “Spawn and Synchronize Execution of RTOS Task” on page 5-14 for more information.

Using the Async Interrupt Block in Simulation and Code Generation

This section describes a *dual-model* approach to the development and implementation of real-time systems that include ISRs. In this approach, you develop one model that includes a plant and a controller for simulation, and another model that only includes the controller for code generation. Using a Simulink library, you can implement changes to both models simultaneously. The next figure shows how changes made to the plant or controller, both of which are in a library, are propagated to the models.

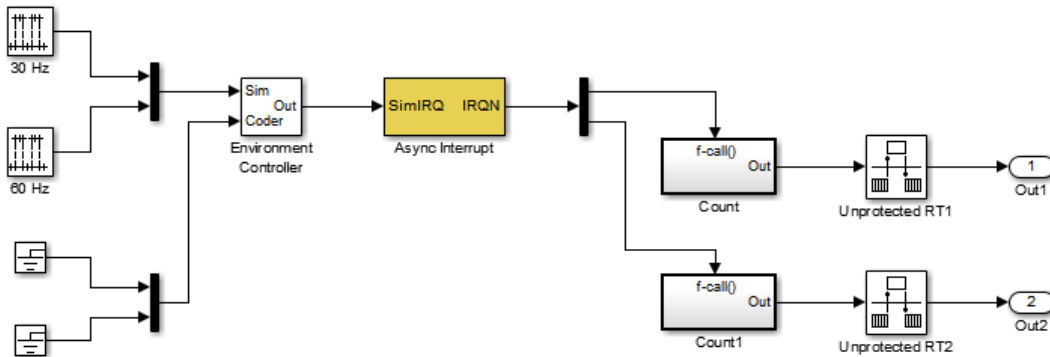


Dual-Model Use of Async Interrupt Block for Simulation and Code Generation

A *single-model* approach is also possible. In this approach, the Plant component of the model is active only in simulation. During code generation, the Plant components are effectively switched out of the system and code is generated only for the interrupt block and controller parts of the model. For an example of this approach, see the `rtwdemo_async` model.

Dual-Model Approach: Simulation

The following block diagram shows a simple model that illustrates the dual-model approach to modeling. During simulation, the Pulse Generator blocks provide simulated interrupt signals.



The simulated interrupt signals are routed through the Async Interrupt block's input port. Upon receiving a simulated interrupt, the block calls the connected subsystem.

During simulation, subsystems connected to Async Interrupt block outputs are executed in order of their priority in the example RTOS (VxWorks). In the event that two or more interrupt signals occur simultaneously, the Async Interrupt block executes the downstream systems in the order specified by their interrupt levels (level 7 gets the highest priority). The first input element maps to the first output element.

You can also use the Async Interrupt block in a simulation without enabling the simulation input. In such a case, the Async Interrupt block inherits the base rate of the model and calls the connected subsystems in order of their priorities in the RTOS. (In this case, the Async Interrupt block behaves as if all inputs received a 1 simultaneously.)

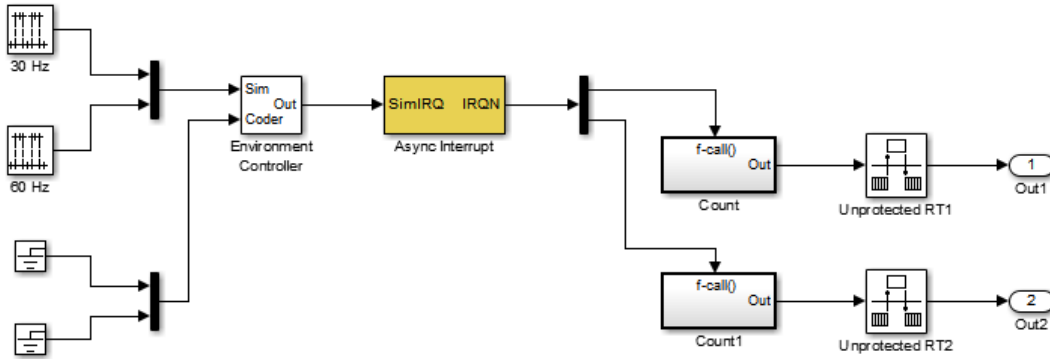
Dual-Model Approach: Code Generation

In the generated code for the sample model,

- Ground blocks provide input signals to the Environment Controller block
- The Async Interrupt block does not use its simulation input

The Ground blocks drive control input of the Environment Controller block, so code is not generated for that signal path. The code generator does not produce code for blocks that drive the simulation control input to the Environment Controller block because that path is not selected during code generation. However, the sample times of driving blocks

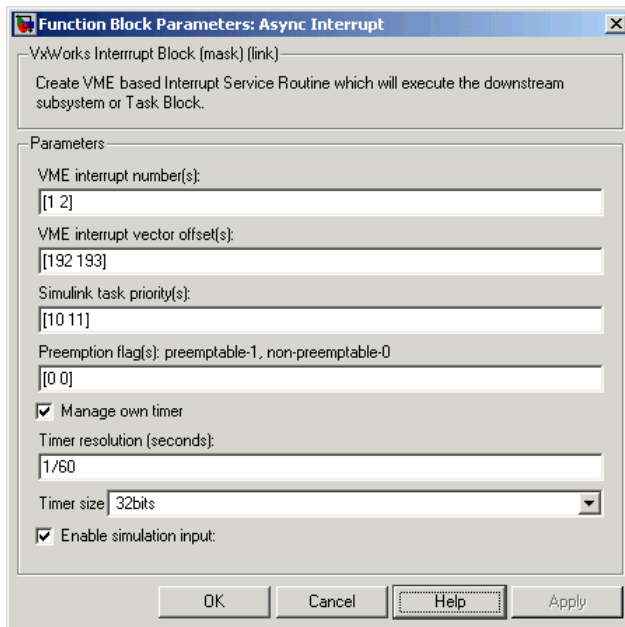
for the simulation input to the Environment Controller block contribute to the sample times supported in the generated code. To avoid including unnecessary sample times in the generated code, use the sample times of the blocks driving the simulation input in the model where generated code is intended.



Standalone functions are installed as ISRs and the interrupt vector table is as follows:

Offset	
192	<code>&isr_num1_vec192()</code>
193	<code>&isr_num2_vec193()</code>

Consider the code generated from this model, assuming that the Async Interrupt block parameters are configured as shown in the next figure.



Initialization Code

In the generated code, the Async Interrupt block installs the code in the Subsystem blocks as interrupt service routines. The interrupt vectors for IRQ1 and IRQ2 are stored at locations 192 and 193 relative to the base of the interrupt vector table, as specified by the **VME interrupt vector offset(s)** parameter.

Installing an ISR requires two RTOS (VxWorks) calls, `int_connect` and `sysInt_Enable`. The Async Interrupt block inserts these calls in the `model_initialize` function, as shown in the following code excerpt.

```
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
/* Connect and enable ISR function: isr_num1_vec192 */
if( intConnect(INUM_TO_IVEC(192), isr_num1_vec192, 0) != OK) {
    printf("intConnect failed for ISR 1.\n");
}
sysIntEnable(1);

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
/* Connect and enable ISR function: isr_num2_vec193 */
if( intConnect(INUM_TO_IVEC(193), isr_num2_vec193, 0) != OK)
{
    printf("intConnect failed for ISR 2.\n");
}
```

```
sysIntEnable(2);
```

The hardware that generates the interrupt is not configured by the Async Interrupt block. Typically, the interrupt source is a VME I/O board, which generates interrupts for specific events (for example, end of A/D conversion). The VME interrupt level and vector are set up in registers or by using jumpers on the board. You can use the `mdlStart` routine of a user-written device driver (S-function) to set up the registers and enable interrupt generation on the board. You must match the interrupt level and vector specified in the Async Interrupt block dialog to the level and vector set up on the I/O board.

Generated ISR Code

The actual ISR generated for IRQ1 in the RTOS (VxWorks) is listed below.

```
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */  
  
void isr_num1_vec192(void)  
{  
    int_T lock;  
    FP_CONTEXT context;  
  
    /* Use tickGet() as a portable tick counter example.  
       A much higher resolution can be achieved with a  
       hardware counter */  
    Async_Code_M->Timing.clockTick2 = tickGet();  
  
    /* disable interrupts (system is configured as non-ive) */  
    lock = intLock();  
  
    /* save floating point context */  
    fppSave(&context);  
  
    /* Call the system: <Root>/Subsystem A */  
    Count(0, 0);  
  
    /* restore floating point context */  
    fppRestore(&context);  
  
    /* re-enable interrupts */  
    intUnlock(lock);  
}
```

There are several features of the ISR that should be noted:

- Because of the setting of the **Preemption Flag(s)** parameter, this ISR is locked; that is, it cannot be preempted by a higher priority interrupt. The ISR is locked and unlocked in the example RTOS (VxWorks) by the `int_lock` and `int_unlock` functions.
- The connected subsystem, `Count`, is called from within the ISR.
- The `Count` function executes algorithmic (model) code. Therefore, the floating-point context is saved and restored across the call to `Count`.
- The ISR maintains its own absolute time counter, which is distinct from other periodic base rate or subrate counters in the system. Timing data is maintained for the use of any blocks executed within the ISR that require absolute or elapsed time.

See “Timers in Asynchronous Tasks” on page 5-42 for details.

Model Termination Code

The model's termination function disables the interrupts in the RTOS (VxWorks):

```
/* Model terminate function */
void Async_Code_terminate(void)
{
    /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Disable interrupt for ISR system: isr_num1_vec192 */
    sysIntDisable(1);

    /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Disable interrupt for ISR system: isr_num2_vec193 */
    sysIntDisable(2);
}
```

More About

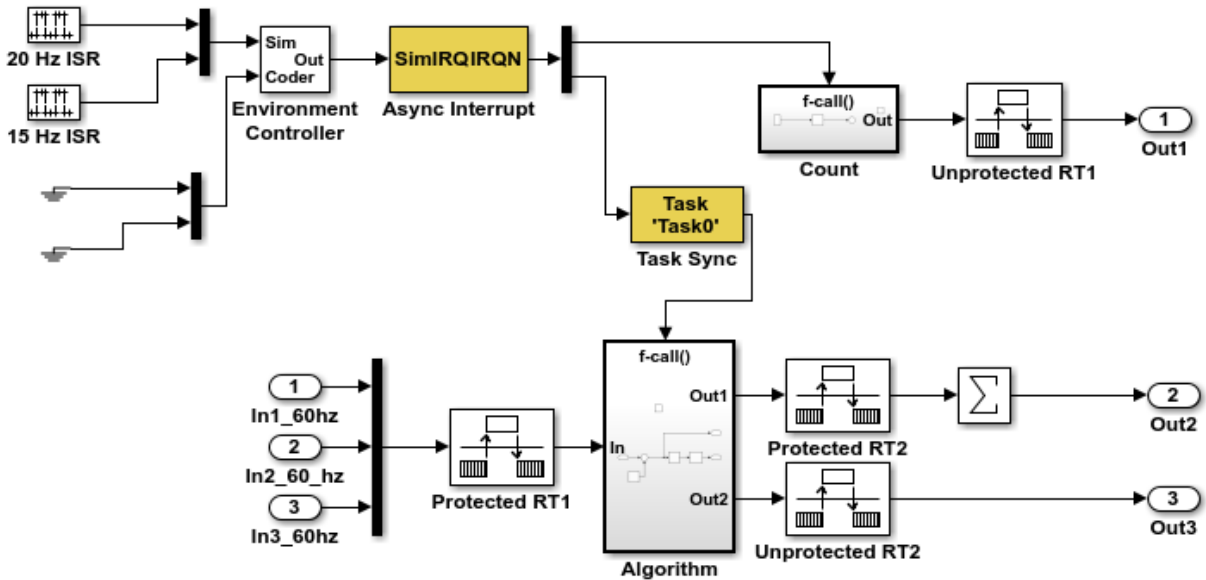
- “Spawn and Synchronize Execution of RTOS Task” on page 5-14
- “Pass Asynchronous Events in RTOS as Input To a Referenced Model” on page 5-30
- “Import Asynchronous Event Data for Simulation” on page 5-53
- “Rate Transitions and Asynchronous Blocks” on page 5-37

Spawn and Synchronize Execution of RTOS Task

This example shows how to simulate and generate code for asynchronous events on a multitasking real-time operating system (VxWorks®). The model shows different techniques for handling asynchronous events depending on the size of the triggered subsystems.

About the Example Model

Open the example model `rtwdemo_async`.



This model shows how to simulate and generate code for asynchronous events on a real-time multitasking system. This model contains two asynchronously executed subsystems, "Count" and "Algorithm." "Count" is executed at interrupt level, whereas "Algorithm" is executed in an asynchronous task. The code generated for these blocks is specifically tailored for the VxWorks operating system. However, you can modify the Async Interrupt and Task Sync blocks to generated code specific to your environment whether you are using an operating system or not.

Generate Code Using Simulink Coder (double-click)	Generate Code Using Embedded Coder (double-click)	Data Transfer Assumptions ...	Display Sample Time Colors (double-click)
--	--	--	--

Copyright 1994-2012 The MathWorks, Inc.

The model simulates an interrupt source and includes an Async Interrupt block, a Task Sync block, function-call subsystems Count and Algorithm, and Rate Transition blocks. The Async Interrupt block creates two Versa Module Eurocard (VME) interrupt service routines (ISRs) that pass interrupt signals to subsystem Count and the Task Sync block. You can place an Async Interrupt block between a simulated interrupt source and one of the following:

- Function call subsystem
- Task Sync block
- A Stateflow® chart configured for a function call input event
- A referenced model with an Inport block that connects to one of the preceding model elements

The Async Interrupt and Task Sync blocks enable the subsystems to execute asynchronously.

Count represents a simple interrupt service routine (ISR) that executes at interrupt level. It is best to keep ISRs as simple as possible. This subsystem includes only a Discrete-Time Integrator block.

Algorithm includes more substance. It includes multiple blocks and produces two output values. Execution of larger subsystems at interrupt level can significantly impact response time for interrupts of equal and lower priority in the system. A better solution for larger subsystems is to use the Task Sync block to represent the ISR for the function-call subsystem.

The Async Interrupt block generates calls to ISRs. Place the block between a simulated interrupt source and one of the following:

- Function call subsystem
- Task Sync block
- A Stateflow® chart configured for a function call input event

For each specified interrupt level, the block generates a Versa Module Eurocard (VME) ISR that executes the connected subsystem, Task Sync block, or chart.

In the example model, the Async Interrupt block is configured for VME interrupts 1 and 2, by using interrupt vector offsets 192 and 193. Interrupt 1 connects directly to subsystem **Count**. Interrupt 2 connects to a Task Sync block, which serves as the ISR for **Algorithm**. Place a Task Sync block in one of the following locations:

- Between an Async Interrupt block and a function-call subsystem or Stateflow® chart.
- At the output port of a Stateflow® chart that has an event, **Output to Simulink**, that you configure as a function call.

In the example model, the Task Sync block is between the Async Interrupt block and function-call subsystem **Algorithm**. The Task Sync block is configured with the task name **Task()**, a priority of 50, a stack size of 8192, and data transfers of the task

synchronized with the caller task. The spawned task uses a semaphore to synchronize task execution. The Async Interrupt block triggers a release of the task semaphore.

Four Rate Transition blocks handle data transfers between ports that operate at different rates. In two instances, Protected Rate Transition blocks protect data transfers (prevent them from being preempted and corrupted). In the other two instances, Unprotected Rate Transition blocks introduce no special behavior. Their presence informs Simulink® of a rate transition.

The code generated for the Async Interrupt and Task Sync blocks is tailored for the example RTOS (VxWorks®). However, you can modify the blocks to generate code specific to your run-time environment.

Data Transfer Assumptions

- Data transfers occur between one reading task and one writing task.
- A read or write operation on a byte-size variable is atomic.
- When two tasks interact, only one can preempt the other.
- For periodic tasks, the task with the faster rate has higher priority than the task with the slower rate. The task with the faster rate preempts the tasks with slower rates.
- Tasks run on a single processor. Time slicing is not allowed.
- Processes do not stop and restart, especially while data is being transferred between tasks.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for input and output appear red and green, respectively. Constants are reddish-blue. Asynchronous interrupts and tasks are purple. The Rate Transition Blocks, which are a hybrid rate (their input and output sample times can differ), are yellow.

Generate Code and Report

Generate code and a code generation report for the model. Generated code for the Async Interrupt and Task Sync blocks is for the example RTOS (VxWorks®). However, you can modify the blocks to generate code for another run-time environment.

1. Create a temporary folder for the build and inspection process.
2. Build the model.

```
### Starting build procedure for model: rtwdemo_async
Warning: Simulink Coder: The tornado.tlc target will be removed in a future release.

### Wrapping unrecognized make command (angle brackets added)
### <make>
### in default batch file
### Successful completion of code generation for model: rtwdemo_async
```

Review Initialization Code

Open the generated source file `rtwdemo_async.c`. The initialization code:

1. Creates and initializes the synchronization semaphore `Task0_semaphore`.

```
*(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID = semBCreate(SEM_Q_PRIORITY,
SEM_EMPTY);
if (rtwdemo_async_DW.SFunction_PWORK.SemID == NULL) {
    printf("semBCreate call failed for block Task0.\n");
}
```

2. Spawns task `task0` and assigns the task priority 50.

```
rtwdemo_async_DW.SFunction_IWORK.TaskID = taskSpawn("Task0",
50.0,
VX_FP_TASK,
8192.0,
(FUNCPTR)Task0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
if (rtwdemo_async_DW.SFunction_IWORK.TaskID == ERROR) {
    printf("taskSpawn call failed for block Task0.\n");
}
```

3. Connects and enables ISR `isr_num1_vec192` for interrupt 1 and ISR `isr_num2_vec193` for interrupt 2.

```
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
/* Connect and enable ISR function: isr_num1_vec192 */
if (intConnect(INUM_TO_IVEC(192), isr_num1_vec192, 0) != OK) {
    printf("intConnect failed for ISR 1.\n");
}

sysIntEnable(1);
```

```

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
/* Connect and enable ISR function: isr_num2_vec193 */
if (intConnect(INUM_TO_IVEC(193), isr_num2_vec193, 0) != OK) {
    printf("intConnect failed for ISR 2.\n");
}

sysIntEnable(2);

/* InitializeConditions for RateTransition: '<Root>/Protected RT1' */
memset(&rtwdemo_async_DW.ProtectedRT1_Buffer[0], 0, 60U * sizeof(real_T));

/* InitializeConditions for RateTransition: '<Root>/Protected RT2' */
memset(&rtwdemo_async_DW.ProtectedRT2_Buffer[0], 0, 60U * sizeof(real_T));

/* SystemInitialize for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates
 * SystemInitialize for SubSystem: '<Root>/Count'
 */
/* System initialize for function-call system: '<Root>/Count' */
rtwdemo_async_DW.Count_PREV_T = rtwdemo_async_M->Timing.clockTick2;

/* InitializeConditions for DiscreteIntegrator: '<S2>/Integrator' */
rtwdemo_async_DW.Integrator_DSTATE_1 = 0.0;

/* SystemInitialize for Output: '<Root>/Out1' incorporates:
 * SystemInitialize for Output: '<S2>/Out'
 */
rtwdemo_async_Y.Out1 = 0.0;

/* SystemInitialize for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates
 * SystemInitialize for SubSystem: '<S4>/Subsystem'
 */

/* System initialize for function-call system: '<S4>/Subsystem' */

/* SystemInitialize for S-Function (vxtask1): '<S5>/S-Function' incorporates:
 * SystemInitialize for SubSystem: '<Root>/Algorithm'
 */

/* System initialize for function-call system: '<Root>/Algorithm' */
rtwdemo_async_M->Timing.clockTick4 = rtwdemo_async_M->Timing.clockTick3;
rtwdemo_async_DW.Algorithm_PREV_T = rtwdemo_async_M->Timing.clockTick4;

/* InitializeConditions for DiscreteIntegrator: '<S1>/Integrator' */
rtwdemo_async_DW.Integrator_DSTATE = 0.0;

```

```

/* SystemInitialize for Outport: '<S1>/Out1' */
memset(&rtwdemo_async_B.Sum[0], 0, 60U * sizeof(real_T));

/* SystemInitialize for Outport: '<Root>/Out3' incorporates:
 * SystemInitialize for Outport: '<S1>/Out2'
 */
rtwdemo_async_Y.Out3 = 0.0;

/* End of SystemInitialize for S-Function (vxtask1): '<S5>/S-Function' */

/* End of SystemInitialize for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}

/* Model terminate function */
static void rtwdemo_async_terminate(void)
{
/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
/* Disable interrupt for ISR system: isr_num1_vec192 */
sysIntDisable(1);

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
/* Disable interrupt for ISR system: isr_num2_vec193 */
sysIntDisable(2);

/* Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates:
 * Terminate for SubSystem: '<S4>/Subsystem'
 */

/* Termination for function-call system: '<S4>/Subsystem' */

/* VxWorks Task Block: '<S5>/S-Function' (vxtask1) */
/* Destroy task: Task0 */
taskDelete(rtwdemo_async_DW.SFunction_IWORK.TaskID);

/* End of Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}

/*=====
 * Start of Classic call interface
 *=====*/
void MdlOutputs(int_T tid)
{
    rtwdemo_async_output(tid);
}

```

```
}

void MdlUpdate(int_T tid)
{
    rtwdemo_async_update(tid);
}

void MdlInitializeSizes(void)
{
}

void MdlInitializeSampleTimes(void)
{
}

void MdlInitialize(void)
{
}

void MdlStart(void)
{
    rtwdemo_async_initialize();
}

void MdlTerminate(void)
{
    rtwdemo_async_terminate();
}

/* Registration function */
RT_MODEL_rtwdemo_async_T *rtwdemo_async(void)
{
    /* Registration code */

    /* initialize non-finites */
    rt_InitInfAndNaN(sizeof(real_T));

    /* initialize real-time model */
    (void) memset((void *)rtwdemo_async_M, 0,
                 sizeof(RT_MODEL_rtwdemo_async_T));

    /* Initialize timing info */
    {
        int_T *mdlTsMap = rtwdemo_async_M->Timing.sampleTimeTaskIDArray;
```

```

mdlTsMap[0] = 0;
mdlTsMap[1] = 1;
rtwdemo_async_M->Timing.sampleTimeTaskIDPtr = (&mdlTsMap[0]);
rtwdemo_async_M->Timing.sampleTimes =
    (&rtwdemo_async_M->Timing.sampleTimesArray[0]);
rtwdemo_async_M->Timing.offsetTimes =
    (&rtwdemo_async_M->Timing.offsetTimesArray[0]);

/* task periods */
rtwdemo_async_M->Timing.sampleTimes[0] = (0.016666666666666666);
rtwdemo_async_M->Timing.sampleTimes[1] = (0.05);

/* task offsets */
rtwdemo_async_M->Timing.offsetTimes[0] = (0.0);
rtwdemo_async_M->Timing.offsetTimes[1] = (0.0);
}

rtmSetTPtr(rtwdemo_async_M, &rtwdemo_async_M->Timing.tArray[0]);

{
    int_T *mdlSampleHits = rtwdemo_async_M->Timing.sampleHitArray;
    int_T *mdlPerTaskSampleHits = rtwdemo_async_M->Timing.perTaskSampleHitsArray;
    rtwdemo_async_M->Timing.perTaskSampleHits = (&mdlPerTaskSampleHits[0]);
    mdlSampleHits[0] = 1;
    rtwdemo_async_M->Timing.sampleHits = (&mdlSampleHits[0]);
}

rtmSetTFinal(rtwdemo_async_M, 0.5);
rtwdemo_async_M->Timing.stepSize0 = 0.016666666666666666;
rtwdemo_async_M->Timing.stepSize1 = 0.05;
rtwdemo_async_M->solverInfoPtr = (&rtwdemo_async_M->solverInfo);
rtwdemo_async_M->Timing.stepSize = (0.016666666666666666);
rtsiSetFixedStepSize(&rtwdemo_async_M->solverInfo, 0.016666666666666666);
rtsiSetSolverMode(&rtwdemo_async_M->solverInfo, SOLVER_MODE_MULTITASKING);

/* block I/O */
rtwdemo_async_M->blockIO = ((void *) &rtwdemo_async_B);
(void) memset(((void *) &rtwdemo_async_B), 0,
              sizeof(B_rtwdemo_async_T));

/* states (dwork) */
rtwdemo_async_M->dwork = ((void *) &rtwdemo_async_DW);
(void) memset(((void *) &rtwdemo_async_DW), 0,
              sizeof(DW_rtwdemo_async_T));

```

```

/* external inputs */
rtwdemo_async_M->inputs = (((void*)&rtwdemo_async_U));
(void)memset((void *)&rtwdemo_async_U, 0, sizeof(ExtU_rtwdemo_async_T));

/* external outputs */
rtwdemo_async_M->outputs = (&rtwdemo_async_Y);
(void)memset((void *)&rtwdemo_async_Y, 0,
             sizeof(ExtY_rtwdemo_async_T));

/* Initialize Sizes */
rtwdemo_async_M->Sizes.numContStates = (0);/* Number of continuous states */
rtwdemo_async_M->Sizes.numY = (3); /* Number of model outputs */
rtwdemo_async_M->Sizes.numU = (60); /* Number of model inputs */
rtwdemo_async_M->Sizes.sysDirFeedThru = (0);/* The model is not direct feedthrough */
rtwdemo_async_M->Sizes.numSampTimes = (2);/* Number of sample times */
rtwdemo_async_M->Sizes.numBlocks = (17);/* Number of blocks */
rtwdemo_async_M->Sizes.numBlockIO = (4);/* Number of block outputs */
return rtwdemo_async_M;
}

/*=====
 * End of Classic call interface
 *=====*/

```

The order of these operations is important. Before the code generator enables the interrupt that activates the task, it must spawn the task.

Review Task and Task Synchronization Code

In the generated source file `rtwdemo_async.c`, review the task and task synchronization code.

The code generator produces the code for function `Task0` from the Task Sync block. That function includes a small amount of interrupt-level code and runs as an RTOS task.

The task waits in an infinite `for` loop until the system releases a synchronization semaphore. If the system releases the semaphore, the function updates its task timer and calls the code generated for the `Algorithm` subsystem.

In the example model, the **Synchronize the data transfer of this task with the caller task** parameter for the Task Sync block is set. This parameter setting updates the timer associated with the Task Sync block (`rtM->Timing.clockTick2`) with the value

of the timer that the Async Interrupt block (`rtM->Timing.clockTick3`) maintains. As a result, code for blocks within the `Algorithm` subsystem use timer values that are based on the time of the most recent interrupt, rather than the most recent activation of `Task0`.

```
{
  /* Wait for semaphore to be released by system: rtdemo_async/Task Sync */
  for (;;) {
    if (semTake(*(SEM_ID *)rtdemo_async_DW.SFunction_PWORK.SemID,NO_WAIT) !=
        ERROR) {
      logMsg("Rate for Task Task0() too fast.\n",0,0,0,0,0,0);

#ifdef STOPONOVERRUN

      logMsg("Aborting real-time simulation.\n",0,0,0,0,0,0);
      semGive(stopSem);
      return(ERROR);

#endif

    } else {
      semTake(*(SEM_ID *)rtdemo_async_DW.SFunction_PWORK.SemID, WAIT_FOREVER);
    }

    /* Use the upstream clock tick counter for this Task. */
    rtdemo_async_M->Timing.clockTick4 = rtdemo_async_M->Timing.clockTick3;

    /* Call the system: '<Root>/Algorithm' */
    {
      {
        int32_T tmp;

        /* RateTransition: '<Root>/Protected RT1' */
        tmp = rtdemo_async_DW.ProtectedRT1_ActiveBufIdx * 60;
        memcpy(&rtdemo_async_B.ProtectedRT1[0],
              &rtdemo_async_DW.ProtectedRT1_Buffer[tmp], 60U * sizeof(real_T));

        /* Output and update for function-call system: '<Root>/Algorithm' */
        {
          real_T tmp;
          int32_T i;
          rtdemo_async_M->Timing.clockTick4 =
            rtdemo_async_M->Timing.clockTick3;

```



```

rtwdemo_async_DW.Algorithm_ELAPS_T =
    rtwdemo_async_M->Timing.clockTick4 -
    rtwdemo_async_DW.Algorithm_PREV_T;
rtwdemo_async_DW.Algorithm_PREV_T = rtwdemo_async_M->Timing.clockTick4;

/* Outport: '<Root>/Out3' incorporates:
 * DiscreteIntegrator: '<S1>/Integrator'
 */
rtwdemo_async_Y.Out3 = rtwdemo_async_DW.Integrator_DSTATE;

/* Sum: '<S1>/Sum' incorporates:
 * Constant: '<S1>/Offset'
 */
for (i = 0; i < 60; i++) {
    rtwdemo_async_B.Sum[i] = rtwdemo_async_B.ProtectedRT1[i] + 1.25;
}

/* End of Sum: '<S1>/Sum' */

/* Sum: '<S1>/Sum1' */
tmp = rtwdemo_async_B.Sum[0];
for (i = 0; i < 59; i++) {
    tmp += rtwdemo_async_B.Sum[i + 1];
}

/* Update for DiscreteIntegrator: '<S1>/Integrator' incorporates:
 * Sum: '<S1>/Sum1'
 */
rtwdemo_async_DW.Integrator_DSTATE += 0.016666666666666666 * (real_T)
    rtwdemo_async_DW.Algorithm_ELAPS_T * tmp;
}
}

{
int32_T i;

/* Update for RateTransition: '<Root>/Protected RT2' */
for (i = 0; i < 60; i++) {
    rtwdemo_async_DW.ProtectedRT2_Buffer[i +
        (rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx == 0) * 60] =
        rtwdemo_async_B.Sum[i];
}

rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx = (int8_T)

```

```

        (rtwdemo_async_DW.ProtectedRT2_ActiveBufIdx == 0);
    /* End of Update for RateTransition: '<Root>/Protected RT2' */
    }
}
}
}
}

```

The code generator produces code for ISRs `isr_num1_vec192` and `isr_num2_vec293`.
ISR `isr_num2_vec192`:

- Disables interrupts.
- Saves floating-point context.
- Calls the code generated for the subsystem that connects to the referenced model Inport block, which receives the interrupt.
- Restores floating-point context.
- Reenables interrupts.

```

void isr_num1_vec192(void)
{
    int_T lock;
    FP_CONTEXT context;

    /* Use tickGet() as a portable tick
       counter example. A much higher resolution can
       be achieved with a hardware counter */
    rtwdemo_async_M->Timing.clockTick2 = tickGet();

    /* disable interrupts (system is configured as non-preemptive) */
    lock = intLock();

    /* save floating point context */
    fppSave(&context);

    /* Call the system: '<Root>/Count' */
    {
        /* Output and update for function-call system: '<Root>/Count' */
        rtwdemo_async_DW.Count_ELAPS_T = rtwdemo_async_M->Timing.clockTick2 -
            rtwdemo_async_DW.Count_PREV_T;
        rtwdemo_async_DW.Count_PREV_T = rtwdemo_async_M->Timing.clockTick2;
    }
}

```

```

/* Output: '<Root>/Out1' incorporates:
 * DiscreteIntegrator: '<S2>/Integrator'
 */
rtwdemo_async_Y.Out1 = rtwdemo_async_DW.Integrator_DSTATE_1;

/* Update for DiscreteIntegrator: '<S2>/Integrator' */
rtwdemo_async_DW.Integrator_DSTATE_1 += 0.016666666666666666 * (real_T)
    rtwdemo_async_DW.Count_ELAPS_T;
}

/* restore floating point context */
fppRestore(&context);

/* re-enable interrupts */
intUnlock(lock);
}

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */

```

ISR `isr_num2_vec293` maintains a timer that stores the tick count at the time that the interrupt occurs. After updating the timer, the ISR releases the semaphore that activates Task0.

```

void isr_num2_vec193(void)
{
    /* Use tickGet() as a portable tick
     counter example. A much higher resolution can
     be achieved with a hardware counter */
    rtwdemo_async_M->Timing.clockTick3 = tickGet();

    /* Call the system: '<S4>/Subsystem' */
    {
        /* Output and update for function-call system: '<S4>/Subsystem' */

        /* VxWorks Task Block: '<S5>/S-Function' (vxtask1) */
        /* Release semaphore for system task: Task0 */
        semGive(*(SEM_ID *)rtwdemo_async_DW.SFunction_PWORK.SemID);
    }
}

/* VxWorks Task Block: '<S5>/S-Function' (vxtask1) */

```

Review Task Termination Code

The Task Sync block generates the following termination code.

```
static void rtwdemo_async_terminate(void)
{
    /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Disable interrupt for ISR system: isr_num1_vec192 */
    sysIntDisable(1);

    /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Disable interrupt for ISR system: isr_num2_vec193 */
    sysIntDisable(2);

    /* Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' incorporates:
     * Terminate for SubSystem: '<S4>/Subsystem'
     */

    /* Termination for function-call system: '<S4>/Subsystem' */

    /* VxWorks Task Block: '<S5>/S-Function' (vxtask1) */
    /* Destroy task: Task0 */
    taskDelete(rtwdemo_async_DW.SFunction_IWORK.TaskID);

    /* End of Terminate for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}
```

Related Information

- Async Interrupt
- Task Sync
- “Generate Interrupt Service Routines”
- “Timers in Asynchronous Tasks”
- “Create a Customized Asynchronous Library”
- “Import Asynchronous Event Data for Simulation”
- “Load Data to Root-Level Input Ports”
- “Asynchronous Events”
- “Rate Transitions and Asynchronous Blocks”

- “Asynchronous Support Limitations”

More About

- “Generate Interrupt Service Routines” on page 5-6
- “Pass Asynchronous Events in RTOS as Input To a Referenced Model” on page 5-30
- “Timers in Asynchronous Tasks” on page 5-42
- “Import Asynchronous Event Data for Simulation” on page 5-53
- “Rate Transitions and Asynchronous Blocks” on page 5-37

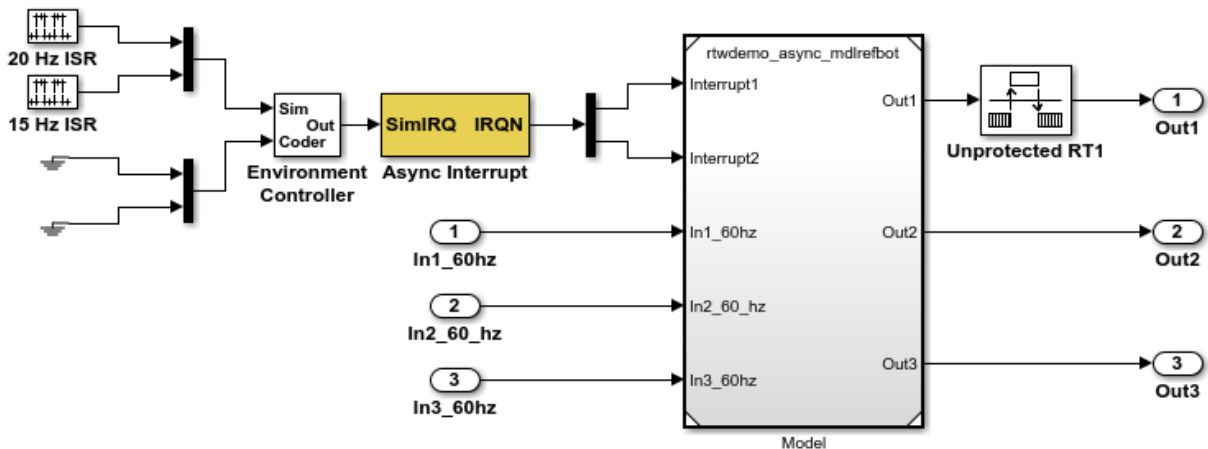
Pass Asynchronous Events in RTOS as Input To a Referenced Model

This example shows how to simulate and generate code for a model that triggers asynchronous events in an example RTOS (VxWorks®) that get passed as input to a referenced model.

Open Example Model

Open the example model `rtwdemo_async_md1refbot`.

Warning: Undefined function 'LibraryBrowserCustomizer' for input arguments of type 'DASudio.CustomizationManager'.



This model shows how to simulate and generate code for asynchronous events on a real-time multitasking system. The two asynchronous events, "Interrupt1" and "Interrupt2", are executed in the referenced model via two different function-call input ports. The code generated for these blocks is specifically tailored for the VxWorks operating system. However, you can modify the Async Interrupt block to generate code specific to your environment whether or not you are using an operating system.

Generate Code Using Simulink Coder (double-click)	Generate Code Using Embedded Coder (double-click)	Data Transfer Assumptions ...	Display Sample Time Colors (double-click)
--	--	--	--

The model simulates an interrupt source and includes an Async Interrupt block and referenced model. The Async Interrupt block creates two Versa Module Eurocard (VME) interrupt service routines (ISRs) that pass interrupt signals to Inport blocks 1 and 2 of the referenced model. You can place an Async Interrupt block between a simulated interrupt source and one of the following:

- Function call subsystem
- Task Sync block
- A Stateflow® chart configured for a function call input event
- A referenced model with a Inport block that connects to one of the preceding model elements

In this example model, the Async Interrupt block passes asynchronous events (function-call trigger signals), `Interrupt1` and `Interrupt2`, to the referenced model through Inport blocks 1 and 2.

The code generated for the Async Interrupt block is tailored for the example real-time operating system (VxWorks). However, you can modify the block to generate code specific to your run-time environment.

Open the referenced model.

The referenced model includes the two Inport blocks that receive the interrupts, each connected to an Asynchronous Task Specification block, function-call subsystems `Count` and `Algorithm`, and Rate Transition blocks. The Asynchronous Task Specification block, in combination with a root-level Inport block, allows a reference model to receive asynchronous function-call input. To use the block:

- 1** Connect the Asynchronous Task Specification block to the output port of a root-level Inport block that outputs a function-call trigger.
- 2** Select the **Output function call** parameter of the Inport block to specify that it accepts function-call signals.
- 3** On the Asynchronous Task Specification parameters dialog box, set the task priority for the asynchronous task associated with an Inport block. Specify an integer or []. If you specify an integer, it must match the priority of the interrupt initiated by the Async Interrupt block in the parent model. If you specify [], the priorities do not have to match.

The Asynchronous Task Specification block for the higher priority interrupt, `interrupt1`, connects to function-call subsystem `Count`. `Count` represents a simple interrupt service routine (ISR). The second Asynchronous Task Specification block

connects to the subsystem `Algorithm`, which includes more substance. It includes multiple blocks and produces two output values. Both subsystems execute at interrupt level.

For each interrupt level specified for the Async Interrupt block in the parent model, the block generates a VME ISR that executes the connected subsystem, Task Sync block, or chart.

In the example top model, the Async Interrupt block is configured for VME interrupts 1 and 2, using interrupt vector offsets 192 and 193. Interrupt 1 is wired to trigger subsystem `Count`. Interrupt 2 is wired to trigger subsystem `Algorithm`.

The Rate Transition blocks handle data transfers between ports that operate at different rates. In two instances, the blocks protect data transfers (prevent them from being preempted and corrupted). In the other instance, no special behavior occurs.

Data Transfer Assumptions

- Data transfers occur between one reading task and one writing task.
- A read or write operation on a byte-sized variable is atomic.
- When two tasks interact, only one can preempt the other.
- For periodic tasks, the task with the faster rate has higher priority than the task with the slower rate. The task with the faster rate preempts the tasks slower rates.
- Tasks run on a single processor. Time slicing is not allowed.
- Processes do not crash and restart, especially while data is being transferred between tasks.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for input and output appear red and green, respectively. Constants are magenta. Asynchronous interrupts are purple. The Rate Transition Blocks, which are hybrid (input and output sample times can differ), appear yellow.

Generate Code and Report

Generate code and a code generation report for the model. Async Interrupt block and Task Sync block generated code is for the example RTOS (VxWorks). However, you can modify the blocks to generate code for another run-time environment.

1. Create a temporary folder for the build and inspection process.

2. Build the model.

Warning: Simulink Coder: The tornado.tlc target will be removed in a future release.

```
### Wrapping unrecognized make command (angle brackets added)
### <make>
### in default batch file
### Successfully updated the model reference RTW target for model: rtwdemo_async_mdltre
### Starting build procedure for model: rtwdemo_async_mdltreftop
Warning: Simulink Coder: The tornado.tlc target will be removed in a future release.

### Wrapping unrecognized make command (angle brackets added)
### <make>
### in default batch file
### Successful completion of code generation for model: rtwdemo_async_mdltreftop
```

Review Initialization Code

Open the generated source file `rtwdemo_async_mdltreftop.c`. The initialization code connects and enables ISR `isr_num1_vec192` for interrupt 1 and ISR `isr_num2_vec193` for interrupt 2.

```
static void rtwdemo_async_mdltreftop_initialize(void)
{
    /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Connect and enable ISR function: isr_num1_vec192 */
    if (intConnect(INUM_TO_IVEC(192), isr_num1_vec192, 0) != OK) {
        printf("intConnect failed for ISR 1.\n");
    }

    sysIntEnable(1);

    /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Connect and enable ISR function: isr_num2_vec193 */
    if (intConnect(INUM_TO_IVEC(193), isr_num2_vec193, 0) != OK) {
        printf("intConnect failed for ISR 2.\n");
    }

    sysIntEnable(2);

    /* SystemInitialize for ModelReference: '<Root>/Model' */
    rtwdemo_async_mdltreftop_Init(&rtwdemo_async_mdltreftop_Y.Out1);

    /* Enable for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
```

```
/* Enable for ModelReference: '<Root>/Model' incorporates:
 * Enable for Inport: '<Root>/In1_60hz'
 * Enable for Inport: '<Root>/In2_60_hz'
 * Enable for Inport: '<Root>/In3_60hz'
 */
rtwdemo_async_mdhrefbot_Interrupt1_Enable();
rtwdemo_async_mdhrefbot_Interrupt2_Enable();

/* End of Enable for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}
```

Review ISR Code

In the generated source file `rtwdemo_async_mdhreftop.c`, review the code for ISRs `isr_num1_vec192` and `isr_num2_vec293`. Each ISR:

- Disables interrupts.
- Saves floating-point context.
- Calls the code generated for the subsystem connected to the referenced model Inport block that receives the interrupt.
- Restores floating-point context.
- Reenables interrupts.

```
void isr_num1_vec192(void)
{
    int_T lock;
    FP_CONTEXT context;

    /* disable interrupts (system is configured as non-preemptive) */
    lock = intLock();

    /* save floating point context */
    fppSave(&context);

    /* Call the system: '<Root>/Model' */
    {
        /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

        /* ModelReference: '<Root>/Model' incorporates:
         * Inport: '<Root>/In1_60hz'
         * Inport: '<Root>/In2_60_hz'
        */
    }
}
```

```

    * Inport: '<Root>/In3_60hz'
    */
    rtwdemo_async_md1refbot_Interrupt1(&rtwdemo_async_md1reftop_Y.Out1);

    /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
}

/* restore floating point context */
fppRestore(&context);

/* re-enable interrupts */
intUnlock(lock);
}

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
void isr_num2_vec193(void)
{
    FP_CONTEXT context;

    /* save floating point context */
    fppSave(&context);

    /* Call the system: '<Root>/Model' */
    {
        /* S-Function (vxinterrupt1): '<Root>/Async Interrupt' */

        /* ModelReference: '<Root>/Model' incorporates:
        * Inport: '<Root>/In1_60hz'
        * Inport: '<Root>/In2_60_hz'
        * Inport: '<Root>/In3_60hz'
        */
        rtwdemo_async_md1refbot_Interrupt2();

        /* End of Outputs for S-Function (vxinterrupt1): '<Root>/Async Interrupt' */
    }

    /* restore floating point context */
    fppRestore(&context);
}

```

Review Task Termination Code

The Task Sync block generates the following termination code.

```
static void rtwdemo_async_mdleftop_terminate(void)
{
    /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Disable interrupt for ISR system: isr_num1_vec192 */
    sysIntDisable(1);

    /* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
    /* Disable interrupt for ISR system: isr_num2_vec193 */
    sysIntDisable(2);
}
```

Related Information

- Async Interrupt
- Asynchronous Task Specification
- “Generate Interrupt Service Routines”
- “Timers in Asynchronous Tasks”
- “Create a Customized Asynchronous Library”
- “Import Asynchronous Event Data for Simulation”
- “Load Data to Root-Level Input Ports”
- “Asynchronous Events”
- “Rate Transitions and Asynchronous Blocks”
- “Asynchronous Support Limitations”

More About

- “Generate Interrupt Service Routines” on page 5-6
- “Spawn and Synchronize Execution of RTOS Task” on page 5-14
- “Timers in Asynchronous Tasks” on page 5-42
- “Import Asynchronous Event Data for Simulation” on page 5-53
- “Rate Transitions and Asynchronous Blocks” on page 5-37

Rate Transitions and Asynchronous Blocks

Because an asynchronous function call subsystem can preempt or be preempted by other model code, an inconsistency arises when more than one signal element is connected to an asynchronous block. The issue is that signals passed to and from the function call subsystem can be in the process of being written to or read from when the preemption occurs. Thus, some old and some new data is used. This situation can also occur with scalar signals in some cases. For example, if a signal is a double (8 bytes), the read or write operation might require two machine instructions. The following sections describe these issues.

In this section...

“About Rate Transitions and Asynchronous Blocks” on page 5-37

“Handle Rate Transitions for Asynchronous Tasks” on page 5-39

“Handle Multiple Asynchronous Interrupts” on page 5-39

Note: The operating system integration techniques that are demonstrated in this section use one or more blocks the blocks in the `vxlib1` on page 5-2 library. These blocks provide starting point examples to help you develop custom blocks for your target environment.

About Rate Transitions and Asynchronous Blocks

The Simulink Rate Transition block is designed to deal with preemption problems that occur in data transfer between blocks running at different rates. These issues are discussed in “Time-Based Scheduling and Code Generation” on page 4-2.

You can handle rate transition issues automatically by selecting the **Automatically handle data transfers between tasks** option on the **Solver** pane of the Configuration Parameters dialog box. This saves you from having to manually insert Rate Transition blocks to avoid invalid rate transitions, including invalid *asynchronous-to-periodic* and *asynchronous-to-asynchronous* rate transitions, in multirate models. For asynchronous tasks, the Simulink engine configures inserted blocks for data integrity but not determinism during data transfers.

For asynchronous rate transitions, the Rate Transition block provides data integrity, but cannot provide determinism. Therefore, when you insert Rate Transition blocks

explicitly, you must clear the **Ensure data determinism** check box in the Block Parameters dialog box.

When you insert a Rate Transition block between two blocks to maintain data integrity and priorities are assigned to the tasks associated with the blocks, the code generator assumes that the higher priority task can preempt the lower priority task and the lower priority task cannot preempt the higher priority task. If the priority associated with task for either block is not assigned or the priorities of the tasks for both blocks are the same, the code generator assumes that either task can preempt the other task.

Priorities of periodic tasks are assigned by the Simulink engine, in accordance with the options specified in the **Solver options** section of the **Solver** pane of the Configuration Parameters dialog box. When the **Periodic sample time constraint** option field of **Solver options** is set to **Unconstrained**, the model base rate priority is set to 40. Priorities for subrates then increment or decrement by 1 from the base rate priority, depending on the setting of the **Higher priority value indicates higher task priority option**.

You can assign priorities manually by using the **Periodic sample time properties** field. The Simulink engine does not assign a priority to asynchronous blocks. For example, the priority of a function call subsystem that connects back to an Async Interrupt block is assigned by the Async Interrupt block.

The **Simulink task priority** field of the Async Interrupt block specifies a priority level (required) for every interrupt number entered in the **VME interrupt number(s)** field. The priority array sets the priorities of the subsystems connected to each interrupt.

For the Task Sync block, if the example RTOS (VxWorks) is the target, the **Higher priority value indicates higher task priority** option should be deselected. The **Simulink task priority** field specifies the block priority relative to connected blocks (in addition to assigning an RTOS priority to the generated task code).

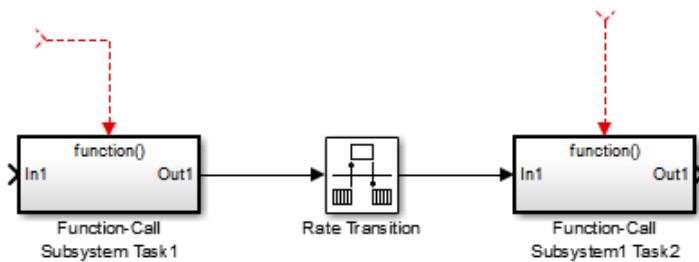
The `vxlib1` library provides two types of rate transition blocks as a convenience. These are simply preconfigured instances of the built-in Simulink Rate Transition block:

- Protected Rate Transition block: Rate Transition block that is configured with the **Ensure data integrity during data transfers** on and **Ensure deterministic data transfer** off.
- Unprotected Rate Transition block: Rate Transition block that is configured with the **Ensure data integrity during data transfers** option off.

Handle Rate Transitions for Asynchronous Tasks

For rate transitions that involve asynchronous tasks, you can maintain data integrity. However, you cannot achieve determinism. You have the option of using the Rate Transition block or target-specific rate transition blocks.

Consider the following model, which includes a Rate Transition block.



You can use the Rate Transition block in either of the following modes:

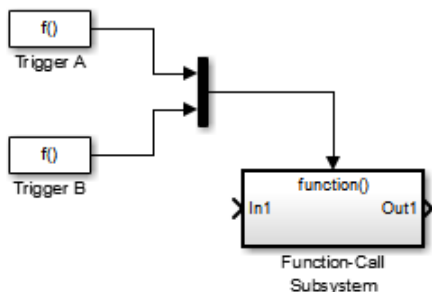
- Maintain data integrity, no determinism
- Unprotected

Alternatively, you can use target-specific rate transition blocks. The following blocks are available for the example RTOS (VxWorks):

- Protected Rate Transition block (reader)
- Protected Rate Transition block (writer)
- Unprotected Rate Transition block

Handle Multiple Asynchronous Interrupts

Consider the following model, in which two functions trigger the same subsystem.



The two tasks must have equal priorities. When priorities are the same, the outcome depends on whether they are firing periodically or asynchronously, and also on a diagnostic setting. The following table and notes describe these outcomes:

Supported Sample Time and Priority for Function Call Subsystem with Multiple Triggers

	Async Priority = 1	Async Priority = 2	Async Priority Unspecified	Periodic Priority = 1	Periodic Priority = 2
Async Priority = 1	Supported (1)				
Async Priority = 2		Supported (1)			
Async Priority Unspecified			Supported (2)		
Periodic Priority = 1				Supported	
Periodic Priority = 2					Supported

- 1 Control these outcomes using the **Tasks with equal priority** option in the **Diagnostics** pane of the Configuration Parameters dialog box; set this diagnostic to **none** if tasks of equal priority cannot preempt each other in the target system.
- 2 For this case, the following warning message is issued unconditionally:

The function call subsystem <name> has multiple asynchronous triggers that do not specify priority. Data integrity will not be maintained if these triggers can preempt one another.

Empty cells in the above table represent multiple triggers with differing priorities, which are unsupported.

The code generator provides absolute time management for a function call subsystem connected to multiple interrupts in the case where timer settings for `TriggerA` and `TriggerB` (time source, resolution) are the same.

Assume that all of the following conditions are true for the model shown above:

- A function call subsystem is triggered by two asynchronous triggers (`TriggerA` and `TriggerB`) having identical priority settings.
- Each trigger sets the source of time and timer attributes by calling the functions `ssSetTimeSource` and `ssSetAsyncTimerAttributes`.
- The triggered subsystem contains a block that needs elapsed or absolute time (for example, a Discrete Time Integrator).

The asynchronous function call subsystem has one global variable, `clockTick#` (where `#` is the task ID associated with the subsystem). This variable stores absolute time for the asynchronous task. There are two ways timing can be handled:

- If the time source is set to `SS_TIMESOURCE_BASERATE`, the code generator produces timer code in the function call subsystem, updating the clock tick variable from the base rate clock tick. Data integrity is maintained if the same priority is assigned to `TriggerA` and `TriggerB`.
- If the time source is `SS_TIMESOURCE_SELF`, generated code for both `TriggerA` and `TriggerB` updates the same clock tick variable from the hardware clock.

The word size of the clock tick variable can be set directly or be established according to the **Application lifespan (days)** setting and the timer resolution set by the `TriggerA` and `TriggerB` S-functions (which must be the same). See “Timers in Asynchronous Tasks” on page 5-42 and “Control Memory Allocation for Time Counters” on page 31-15 for more information.

More About

- “Time-Based Scheduling and Code Generation” on page 4-2
- “Asynchronous Support Limitations” on page 5-57

Timers in Asynchronous Tasks

An ISR can set a source for absolute time. This is done with the function `ssSetTimeSource`. The function `ssSetTimeSource` cannot be called before `ssSetOutputPortWidth` is called. If this occurs, the program will come to a halt and generate an error message. `ssSetTimeSource` has the following three options:

- `SS_TIMESOURCE_SELF`: Each generated ISR maintains its own absolute time counter, which is distinct from a periodic base rate or subrate counters in the system. The counter value and the timer resolution value (specified in the **Timer resolution (seconds)** parameter of the Async Interrupt block) are used by downstream blocks to determine absolute time values required by block computations.
- `SS_TIMESOURCE_CALLER`: The ISR reads time from a counter maintained by its caller. Time resolution is thus the same as its caller's resolution.
- `SS_TIMESOURCE_BASERATE`: The ISR can read absolute time from the model's periodic base rate. Time resolution is thus the same as its base rate resolution.

Note: The operating system integration techniques that are demonstrated in this section use one or more blocks the blocks in the `vxlib1` on page 5-2 library. These blocks provide starting point examples to help you develop custom blocks for your target environment.

By default, the counter is implemented as a 32-bit unsigned integer member of the `Timing` substructure of the real-time model structure. For a target that supports the `rtModel` data structure, when the time data type is not set by using `ssSetAsyncTimeDataType`, the counter word size is determined by the **Application lifespan (days)** model parameter. As an example (from ERT target code),

```
/* Real-time Model Data Structure */
struct _RT_MODEL_elapseTime_exp_Tag {
    const char *errorStatus;

    /*
     * Timing:
     * The following substructure contains information regarding
     * the timing information for the model.
     */
    struct {
        uint32_T clockTick1;
        uint32_T clockTick2;
    };
};
```

```

    } Timing;
};

```

The example omits unused fields in the `Timing` data structure (a feature of ERT target code not found in GRT). For a target that supports the `rtModel` data structure, the counter word size is determined by the **Application lifespan (days)** model parameter.

By default, the `vxlib1` library blocks for the example RTOS (VxWorks) set the timer source to `SS_TIMESOURCE_SELF` and update their counters by using the system call `tickGet`. `tickGet` returns a timer value maintained by the RTOS kernel. The maximum word size for the timer is `UINT32`. The following example shows a generated call to `tickGet`.

```

/* VxWorks Interrupt Block: '<Root>/Async Interrupt' */
void isr_num2_vec193(void)
{
    /* Use tickGet() as a portable tick counter example. A much
       higher resolution can be achieved with a hardware counter */
    rtM->Timing.clockTick2 = tickGet();
    . . .
}

```

The `tickGet` call is supplied only as an example. It can (and in many instances should) be replaced by a timing source that has better resolution. If you are implementing a custom asynchronous block for an RTOS other than the example RTOS (VxWorks), you should either generate an equivalent call to the target RTOS, or generate code to read a timer register on the target hardware.

The default **Timer resolution (seconds)** parameter of your Async Interrupt block implementation should be changed to match the resolution of your target's timing source.

The counter is updated at interrupt level. Its value represents the tick value of the timing source at the most recent execution of the ISR. The rate of this timing source is unrelated to sample rates in the model. In fact, typically it is faster than the model's base rate. Select the timer source and set its rate and resolution based on the expected rate of interrupts to be serviced by the Async Interrupt block.

For an example of timer code generation, see “Async Interrupt Block Implementation” on page 5-46.

Related Examples

- “Generate Interrupt Service Routines” on page 5-6

- “Spawn and Synchronize Execution of RTOS Task” on page 5-14
- “Timers in Asynchronous Tasks” on page 5-42
- “Create a Customized Asynchronous Library” on page 5-45
- “Import Asynchronous Event Data for Simulation” on page 5-53

More About

- “Absolute and Elapsed Time Computation” on page 3-2
- “Time-Based Scheduling and Code Generation” on page 4-2
- “Asynchronous Events” on page 5-2
- “Asynchronous Support Limitations” on page 5-57

Create a Customized Asynchronous Library

This topic describes how to implement asynchronous blocks for use with your target RTOS, using the Async Interrupt and Task Sync blocks as a starting point. Rate Transition blocks are target-independent, so you do not need to develop customized rate transition blocks. The following sections provide implementation details.

In this section...

“About Implementing Asynchronous Blocks” on page 5-45

“Async Interrupt Block Implementation” on page 5-46

“Task Sync Block Implementation” on page 5-50

“asynclib.tlc Support Library” on page 5-51

Note: The operating system integration techniques that are demonstrated in this section use one or more blocks the blocks in the `vxlib1` on page 5-2 library. These blocks provide starting point examples to help you develop custom blocks for your target environment.

About Implementing Asynchronous Blocks

You can customize the asynchronous library blocks by modifying the block implementation. These files are

- The block's underlying S-function MEX-file
- The TLC files that control code generation of the block

In addition, you need to modify the block masks to remove references specific to the example RTOS (VxWorks) and to incorporate parameters required by your target RTOS.

Custom block implementation is an advanced topic, requiring familiarity with the Simulink MEX S-function format and API, and with the Target Language Compiler (TLC). These topics are covered in the following documents:

- Simulink topics “What Is an S-Function?”, “Use S-Functions in Models”, “How S-Functions Work”, and “Implementing S-Functions” describe MEX S-functions and the S-function API in general.
- The “Inlining S-Functions”, “Inline C MEX S-Functions”, and “S-Functions and Code Generation” on page 24-99 describe how to create a TLC block implementation for use in code generation.

The following sections discuss the C/C++ and TLC implementations of the asynchronous library blocks, including required `SimStruct` macros and functions in the TLC asynchronous support library (`asynclib.tlc`).

Async Interrupt Block Implementation

The source files for the Async Interrupt block are located in `matlabroot/rtw/c/tornado/devices` (open):

- `vxinterrupt1.c`: C MEX-file source code, for use in configuration and simulation
- `vxinterrupt1.tlc`: TLC implementation, for use in code generation
- `asynclib.tlc`: library of TLC support functions, called by the TLC implementation of the block. The library calls are summarized in “`asynclib.tlc` Support Library” on page 5-51.

C MEX Block Implementation

Most of the code in `vxinterrupt1.c` performs ordinary functions that are not related to asynchronous support (for example, obtaining and validating parameters from the block mask, marking parameters nontunable, and passing parameter data to the `model.rtw` file).

The `mdlInitializeSizes` function uses special `SimStruct` macros and `SS_OPTIONS` settings that are required for asynchronous blocks, as described below.

Note that the following macros cannot be called before `ssSetOutputPortWidth` is called:

- `ssSetTimeSource`
- `ssSetAsyncTimerAttributes`
- `ssSetAsyncTimerResolutionEl`
- `ssSetAsyncTimerDataType`
- `ssSetAsyncTimerDataTypeEl`
- `ssSetAsyncTaskPriorities`
- `ssSetAsyncTaskPrioritiesEl`

If one of the above macros is called before `ssSetOutputPortWidth`, the following error message appears:

```
SL_SfcnMustSpecifyPortWidthBfCallSomeMacro {
```

```
S-function '%s' in '%<BLOCKFULLPATH>'
must set output port %d width using
ssSetOutputPortWidth before calling macro %s
}
```

ssSetAsyncTimerAttributes

`ssSetAsyncTimerAttributes` declares that the block requires a timer, and sets the resolution of the timer as specified in the **Timer resolution (seconds)** parameter.

The function prototype is

```
ssSetAsyncTimerAttributes(SimStruct *S, double res)
```

where

- `S` is a `Simstruct` pointer.
- `res` is the **Timer resolution (seconds)** parameter value.

The following code excerpt shows the call to `ssSetAsyncTimerAttributes`.

```
/* Setup Async Timer attributes */
ssSetAsyncTimerAttributes(S,mxGetPr(TICK_RES)[0]);
```

ssSetAsyncTaskPriorities

`ssSetAsyncTaskPriorities` sets the Simulink task priority for blocks executing at each interrupt level, as specified in the block's **Simulink task priority** field.

The function prototype is

```
ssSetAsyncTaskPriorities(SimStruct *S, int numISRs,
                        int *priorityArray)
```

where

- `S` is a `SimStruct` pointer.
- `numISRs` is the number of interrupts specified in the **VME interrupt number(s)** parameter.
- `priorityarray` is an integer array containing the interrupt numbers specified in the **VME interrupt number(s)** parameter.

The following code excerpt shows the call to `ssSetAsyncTaskPriorities`:

```
/* Setup Async Task Priorities */
```

```

    priorityArray = malloc(numISRs*sizeof(int_T));
    for (i=0; i<numISRs; i++) {
        priorityArray[i] = (int_T)(mxGetPr(ISR_PRIORITIES)[i]);
    }
    ssSetAsyncTaskPriorities(S, numISRs, priorityArray);
    free(priorityArray);
    priorityArray = NULL;
}

```

SS_OPTION Settings

The code excerpt below shows the SS_OPTION settings for `vxinterrupt1.c`. SS_OPTION_ASYNC_HONOR_INTERRUPT should be used when a function call subsystem is attached to an interrupt. For more information, see the documentation for SS_OPTION and SS_OPTION_ASYNC_HONOR_INTERRUPT in `matlabroot/simulink/include/simstruc.h`.

```

ssSetOptions( S, (SS_OPTION_EXCEPTION_FREE_CODE |
                 SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME |
                 SS_OPTION_ASYNC_HONOR_INTERRUPT |

```

TLC Implementation

This section discusses each function of `vxinterrupt1.tlc`, with an emphasis on target-specific features that you will need to change to generate code for your target RTOS.

Generate #include Directives

`vxinterrupt1.tlc` begins with the statement

```
%include "vxlib.tlc"
```

`vxlib.tlc` is a target-specific file that generates directives to include header files for the example RTOS (VxWorks). You should replace this with a file that generates includes for your target RTOS.

BlockInstanceSetup Function

For each connected output of the Async Interrupt block, `BlockInstanceSetup` defines a function name for the corresponding ISR in the generated code. The functions names are of the form

```
isr_num_vec_offset
```

where *num* is the ISR number defined in the **VME interrupt number(s)** block parameter, and *offset* is an interrupt table offset defined in the **VME interrupt vector offset(s)** block parameter.

In a custom implementation, this naming convention is optional.

The function names are cached for use by the `Outputs` function, which generates the actual ISR code.

Outputs Function

`Outputs` iterates over the connected outputs of the Async Interrupt block. An ISR is generated for each such output.

The ISR code is cached in the "Functions" section of the generated code. Before generating the ISR, `Outputs` does the following:

- Generates a call to the downstream block (cached in a temporary buffer).
- Determines whether the ISR should be locked or not (as specified in the **Preemption Flag(s)** block parameter).
- Determines whether the block connected to the Async Interrupt block is a Task Sync block. (This information is obtained by using the `asynclib` calls `LibGetFcnCallBlock` and `LibGetBlockAttribute`.) If so,
 - The preemption flag for the ISR must be set to 1. An error results otherwise.
 - The RTOS (VxWorks) calls to save and restore floating-point context are generated, unless the user has configured the model for integer-only code generation.

When generating the ISR code, `Outputs` calls the `asynclib` function `LibNeedAsyncCounter` to determine whether a timer is required by the connected subsystem. If so, and if the time source is set to be `SS_TIMESOURCE_SELF` by `ssSetTimeSource`, `LibSetAsyncCounter` is called to generate an RTOS (VxWorks) `tickGet` function call and update the counter. In your implementation, you should generate either an equivalent call to the target RTOS, or generate code to read the a timer register on the target hardware.

Start Function

The `Start` function generates the required RTOS (VxWorks) calls (`int_connect` and `sysInt_Enable`) to connect and enable each ISR. You should replace this with calls to your target RTOS.

Terminate Function

The `Terminate` function generates the call `sysIntDisable` to disable each ISR. You should replace this with calls to your target RTOS.

Task Sync Block Implementation

The source files for the Task Sync block are located in *matlabroot*/rtw/c/tornado/devices (open). They are

- `vxtask1.cpp`: MEX-file source code, for use in configuration and simulation.
- `vxtask1.tlc`: TLC implementation, for use in code generation.
- `asynclib.tlc`: library of TLC support functions, called by the TLC implementation of the block. The library calls are summarized in “asynclib.tlc Support Library” on page 5-51.

C MEX Block Implementation

Like the Async Interrupt block, the Task Sync block sets up a timer, in this case with a fixed resolution. The priority of the task associated with the block is obtained from the **Simulink task priority** parameter. The `SS_OPTION` settings are the same as those used for the Async Interrupt block.

```
ssSetAsyncTimerAttributes(S, 0.01);

priority = (int_T) (*(mxGetPr(PRIORITY)));
ssSetAsyncTaskPriorities(S,1,&priority);

ssSetOptions(S, (SS_OPTION_EXCEPTION_FREE_CODE |
                 SS_OPTION_ASYNCHRONOUS |
                 SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME |
                )
}
```

TLC Implementation

Generate #include Directives

`vxtask1.tlc` begins with the statement

```
%include "vxlib.tlc"
```

`vxlib.tlc` is a target-specific file that generates directives to include header files for the example RTOS (VxWorks). You should replace this with a file that generates includes for your target RTOS.

BlockInstanceSetup Function

The `BlockInstanceSetup` function derives the task name, block name, and other identifiers used later in code generation. It also checks for and warns about unconnected

block conditions, and generates a storage declaration for a semaphore (`stopSem`) that is used in case of interrupt overflow conditions.

Start Function

The `Start` function generates the required RTOS (VxWorks) calls to define storage for the semaphore that is used in management of the task spawned by the Task Sync block. Depending on the value of the `CodeFormat` TLC variable of the target, either a static storage declaration or a dynamic memory allocation call is generated. This function also creates a semaphore (`semBCreate`) and spawns an RTOS task (`taskSpawn`). You should replace these with calls to your target RTOS.

Outputs Function

The `Outputs` function generates an example RTOS (VxWorks) task that waits for a semaphore. When it obtains the semaphore, it updates the block's tick timer and calls the downstream subsystem code, as described in “Spawn and Synchronize Execution of RTOS Task” on page 5-14. `Outputs` also generates code (called from interrupt level) that grants the semaphore.

Terminate Function

The `Terminate` function generates the example RTOS (VxWorks) call `taskDelete` to end execution of the task spawned by the block. You should replace this with calls to your target RTOS.

Note also that if the target RTOS has dynamically allocated memory associated with the task, the `Terminate` function should deallocate the memory.

asynclib.tlc Support Library

`asynclib.tlc` is a library of TLC functions that support the implementation of asynchronous blocks. Some functions are specifically designed for use in asynchronous blocks. For example, `LibSetAsyncCounter` generates a call to update a timer for an asynchronous block. Other functions are utilities that return information required by asynchronous blocks (for example, information about connected function call subsystems).

The following table summarizes the public calls in the library. For details, see the library source code and the `vxinterrupt1.tlc` and `vxtask1.tlc` files, which call the library functions.

Summary of `asynclib.tlc` Library Functions

Function	Description
<code>LibBlockExecuteFcnCall</code>	For use by inlined S-functions with function call outputs. Generates code to execute a function call subsystem.
<code>LibGetBlockAttribute</code>	Returns a field value from a block record.
<code>LibGetFcnCallBlock</code>	Given an S-Function block and call index, returns the block record for the downstream function call subsystem block.
<code>LibGetCallerClockTickCounter</code>	Provides access to the time counter of an upstream asynchronous task.
<code>LibGetCallerClockTickCounter-HighWord</code>	Provides access to the high word of the time counter of an upstream asynchronous task.
<code>LibManageAsyncCounter</code>	Determines whether an asynchronous task needs a counter and manages its own timer.
<code>LibNeedAsyncCounter</code>	If the calling block requires an asynchronous counter, returns <code>TLC_TRUE</code> , otherwise returns <code>TLC_FALSE</code> .
<code>LibSetAsyncClockTicks</code>	Returns code that sets <code>clockTick</code> counters that are to be maintained by the asynchronous task.
<code>LibSetAsyncCounter</code>	Generates code to set the tick value of the block's asynchronous counter.
<code>LibSetAsyncCounterHighWord</code>	Generates code to set the tick value of the high word of the block's asynchronous counter

More About

- “Asynchronous Events” on page 5-2
- “Generate Interrupt Service Routines” on page 5-6
- “Spawn and Synchronize Execution of RTOS Task” on page 5-14
- “Pass Asynchronous Events in RTOS as Input To a Referenced Model” on page 5-30
- “Timers in Asynchronous Tasks” on page 5-42
- “Import Asynchronous Event Data for Simulation” on page 5-53
- “Rate Transitions and Asynchronous Blocks” on page 5-37
- “Asynchronous Support Limitations” on page 5-57

Import Asynchronous Event Data for Simulation

Capabilities

You can import asynchronous event data into a function-call subsystem via an Inport block. For standalone fixed-step simulations, you can specify:

- The time points at which each asynchronous event occurs
- The number of asynchronous events at each time point

Input Data Format

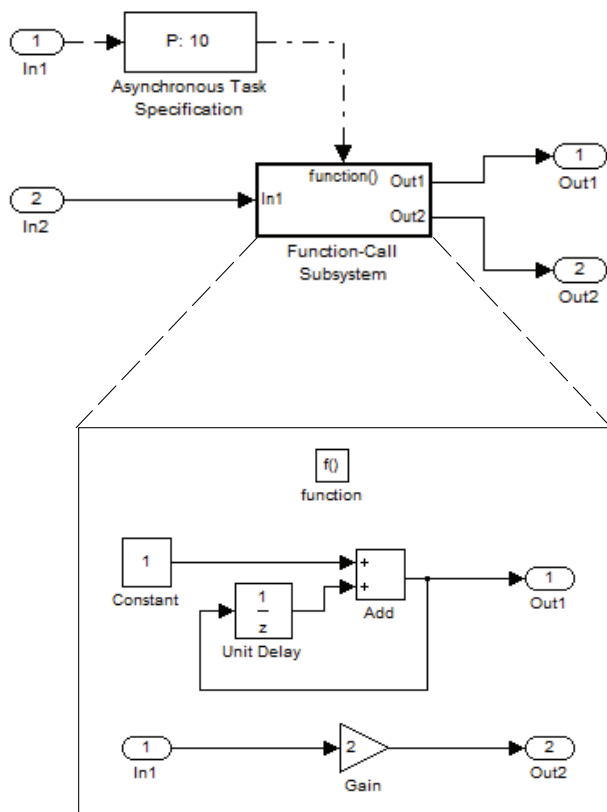
You can enter your asynchronous data at the MATLAB command line or on the **Data Import/Export** pane of the Configuration Parameters dialog box. In either case, a number of restrictions apply to the data format.

- The expression for the parameter **Data Import/Export > Input** must be a comma-separated list of tables.
- The table corresponding to the input port outputting asynchronous events must be a column vector containing time values for the asynchronous events.
 - The time vector of the asynchronous events must be of double data type and monotonically increasing.
 - All time data must be integer multiples of the model step size.
 - To specify multiple function calls at a given time step, you must repeat the time value accordingly. In other words, if you wish to specify three asynchronous events at $t = 1$ and two events at $t = 9$, then you must list 1 three times and 9 twice in your time vector. (`t = [1 1 1 9 9]'`)
- The table corresponding to normal data input port can be of any other supported format.

See “Load Data to Root-Level Input Ports” for more information.

Example

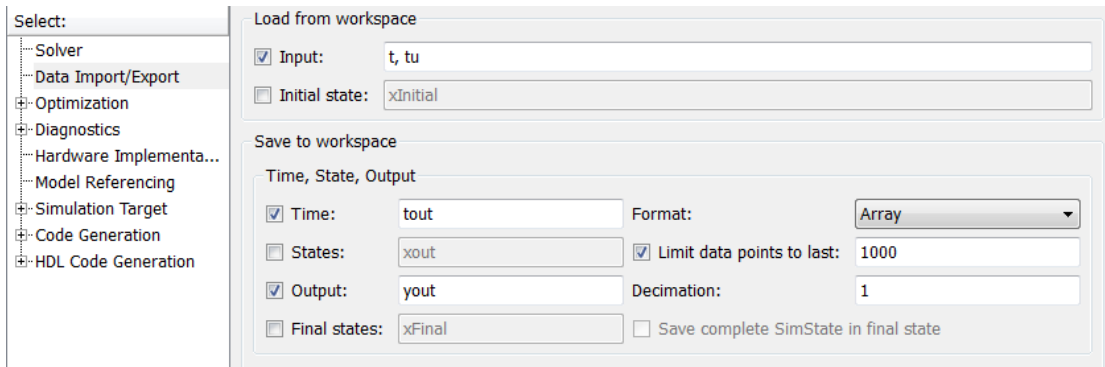
In this model, a function-call subsystem is used to track the total number of asynchronous events and to multiply a set of inputs by 2.



- 1 To input data via the Configuration Parameters dialog box,
 - a Select **Simulation > Configuration Parameters > Data Import/Export**.
 - b Select the **Input** parameter.
 - c For this example, enter the following command in the MATLAB window:


```
>> t = [1 1 5 9 9 9]', u = [[0:10]' [0:10]']
```

 Alternatively, you can enter the data as t , tu in the Data Import/Export pane:



Here, t is a column vector containing the times of asynchronous events for Inport block In1 while tu is a table of input values versus time for Inport block In2.

- 2 By default, the **Time** and **Output** options are selected and the output variables are named $tout$ and $yout$.
- 3 Simulate the model.
- 4 Display the output by entering `[tout yout]` at the MATLAB command line and obtain:

ans =

0	0	-1
1	2	2
2	2	2
3	2	2
4	2	2
5	3	10
6	3	10
7	3	10
8	3	10
9	6	18
10	6	18

Here the first column contains the simulation times.

The second column represents the output of Out1 — the total number of asynchronous events. Since the function-call subsystem is triggered twice at $t = 1$,

the output is 2. It is not called again until $t = 5$, and so does not increase to 3 until then. Finally, it is called three times at 9, so it increases to 6.

The third column contains the output of Out2 obtained by multiplying the input value at each asynchronous event time by 2. At any other time, the output is held at its previous value

More About

- “Asynchronous Events” on page 5-2
- “Pass Asynchronous Events in RTOS as Input To a Referenced Model” on page 5-30
- “Load Data to Root-Level Input Ports”
- “Rate Transitions and Asynchronous Blocks” on page 5-37

Asynchronous Support Limitations

In this section...

“Asynchronous Task Priority” on page 5-57

“Convert an Asynchronous Subsystem into a Model Reference” on page 5-57

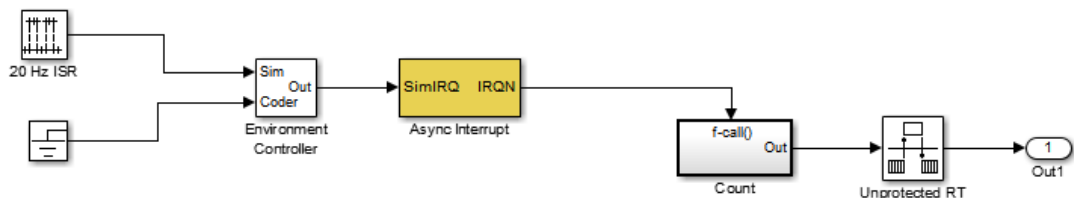
Asynchronous Task Priority

The Simulink product does not simulate asynchronous task behavior. Although you can specify a task priority for an asynchronous task represented in a model with the Task Sync block, the priority setting is for code generation purposes only and is not honored during simulation.

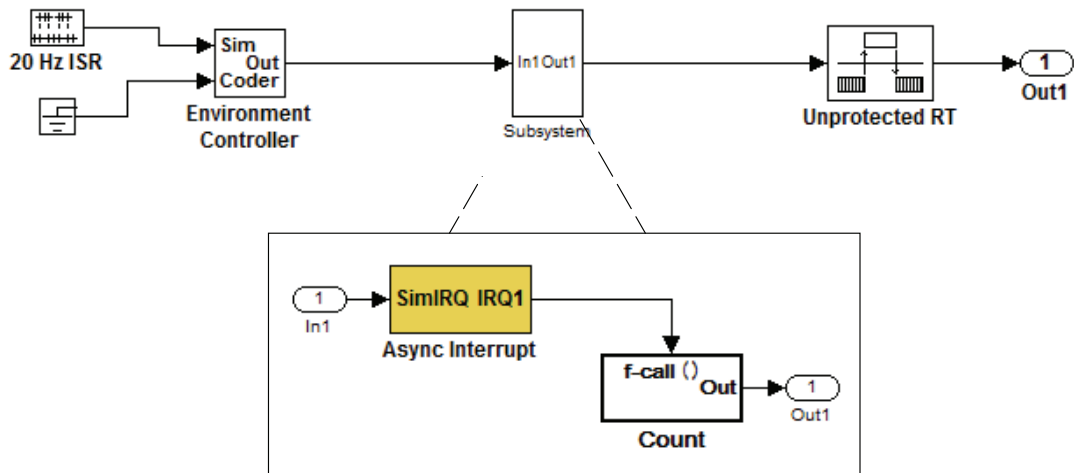
Convert an Asynchronous Subsystem into a Model Reference

You can use the Asynchronous Task Specification block to specify an asynchronous function-call input to a model reference. However, you must convert the Async Interrupt and Function-Call blocks into a subsystem and then convert the subsystem into a model reference.

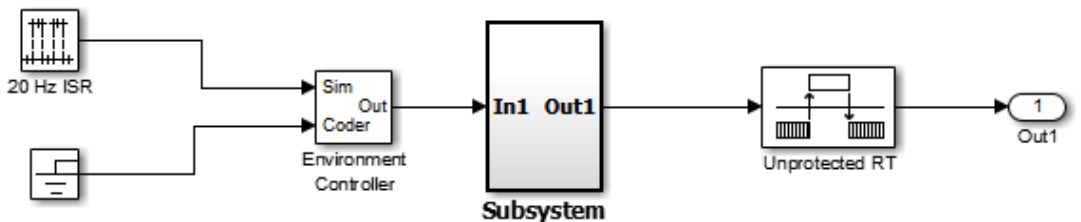
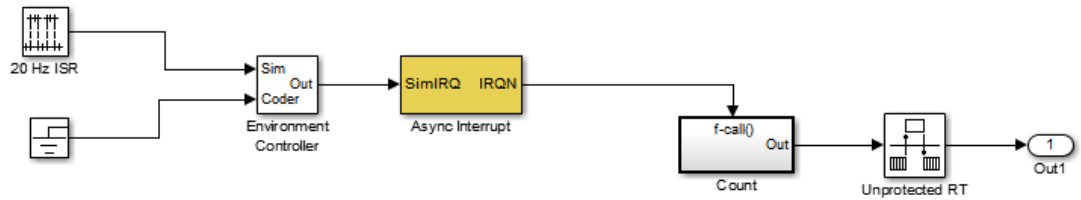
Following is an example with step-by-step instructions for conversion.



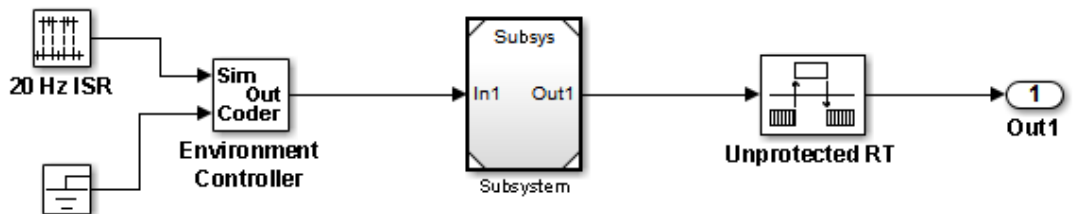
- 1 Convert the Async Interrupt and Count blocks into a subsystem. Select both blocks and right-click Count. From the menu, select **Subsystem & Model Reference > Create Subsystem from Selection**.



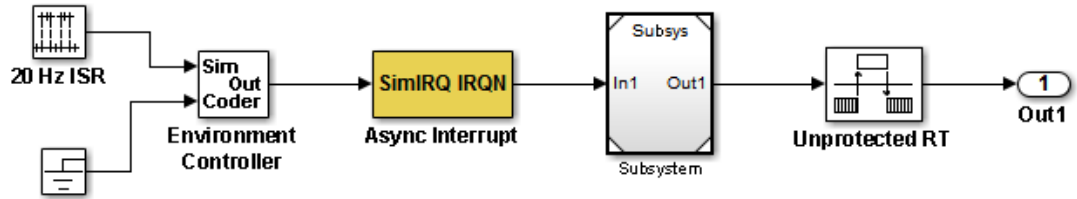
- 2 To prepare for converting the new subsystem to a Model block, set the following configuration parameters in the top model. Open the Configuration Parameters dialog box.
 - Under Diagnostics, navigate to the Sample Time pane. Then set **Multitask rate transition** to **error** and **Multitask conditionally executed subsystem** to **error**.
 - Under Diagnostics, navigate to the Connectivity pane. Set **Mux blocks used to create bus signals**, **Bus signal treated as vector**, and **Invalid function-call connection** to **error**. Also set **Context-dependent inputs** to **Enable All**.
 - Under Diagnostics, navigate to the Data Validity pane and set the **Multitask data store** option to **error**.
 - On the **All Parameters** tab, set **Underspecified initialization detection** to **Simplified**.
 - If your model is large or complex, run the Model Advisor checks in the folder “Migrating to Simplified Initialization Mode Overview” and make the suggested changes.
- 3 Convert the subsystem to an atomic subsystem. Select **Edit > Subsystem Parameters > Treat as atomic unit**.



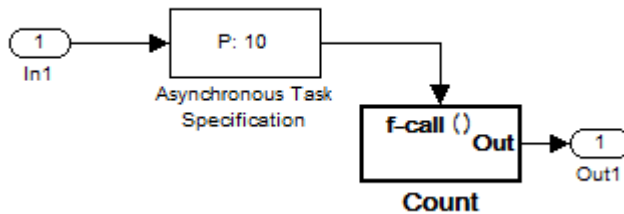
- 4 Convert the subsystem to a Model block. Right-click the subsystem and select **Subsystem & Model Reference > Convert Subsystem to > Referenced Model**. A window opens with a model reference block inside of it.
- 5 Replace the subsystem in the top model with the new model reference block.



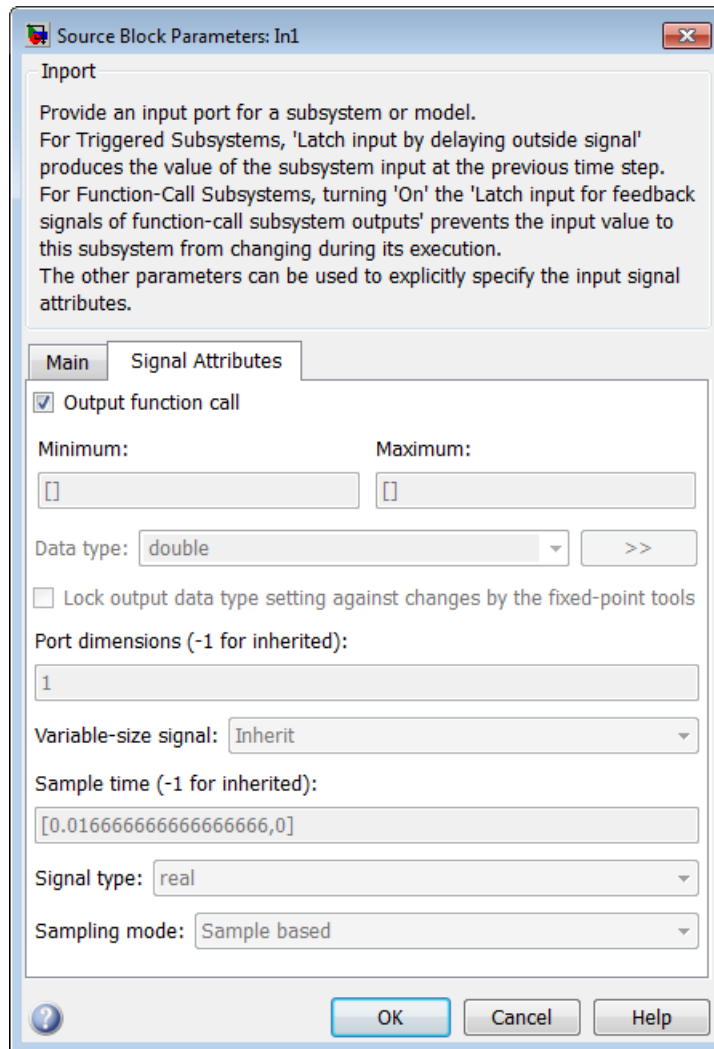
- 6 Move the Async Interrupt block from the model reference to the top model, before the model reference block.



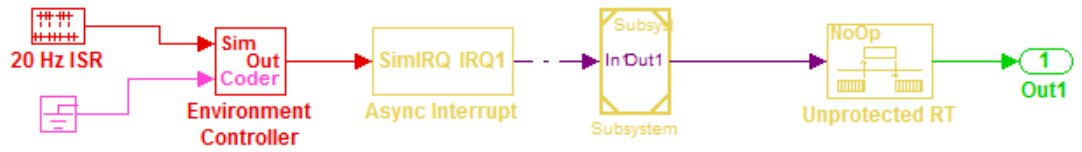
- 7 Insert an Asynchronous Task Specification block in the model reference. Set the priority of the Asynchronous Task Specification block. (For more information on setting the priority, see Asynchronous Task Specification.)



- 8 In the model reference, double-click the input port to open its Source Block Parameters dialog box. Click the **Signal Attributes** tab and select the **Output function call** option. Click **OK**.



- 9 Save your model and then perform **Simulation > Update Diagram** to verify your settings.



More About

- “Asynchronous Events” on page 5-2

Subsystems in Simulink Coder

- “Code Generation of Subsystems” on page 6-2
- “Generate Code and Executables for Individual Subsystem” on page 6-4
- “Inline Subsystem Code” on page 6-7
- “Generate Subsystem Code as Separate Function and Files” on page 6-10
- “Generate Reusable Function for Identical Subsystems Within a Model” on page 6-11
- “Optimize Code for Identical Nested Subsystems” on page 6-14
- “Generate Reusable Code for Subsystems Containing S-Function Blocks” on page 6-15
- “Generate Reusable Code from Stateflow Charts” on page 6-16
- “Code Reuse Limitations for Subsystems” on page 6-17
- “Code Reuse For Subsystems Shared Across Models” on page 6-20
- “Reusable Library Subsystem” on page 6-21
- “Code Generation of Constant Parameters” on page 6-23
- “Shared Constant Parameters for Code Reuse” on page 6-24
- “Generate Reusable Code for Subsystems Shared Across Models” on page 6-28
- “Determine Why Subsystem Code Is Not Reused” on page 6-36

Code Generation of Subsystems

For you to control how code is generated for a nonvirtual subsystem, the code generator provides subsystem parameters that you can use. The categories of nonvirtual subsystems are:

- *Conditionally executed* subsystems. Execution depends upon a control signal or control block. These subsystems include:
 - Triggered
 - Enabled
 - Action
 - Iterator
 - Function-call

For more information, see “Conditional Subsystems”.

- *Atomic* subsystems: A virtual subsystem can be declared atomic (and therefore nonvirtual) by using the “Treat as atomic unit” parameter in the Subsystem Parameters dialog box.

For more information on nonvirtual subsystems and atomic subsystems, see “Systems and Subsystems” and open the Subsystem Semantics library.

You can design and configure your model to control the code generated from nonvirtual subsystems.

To...	See...
Generate inlined code from a selected nonvirtual subsystem.	“Inline Subsystem Code” on page 6-7
Generate code for only a subsystem.	“Generate Code and Executables for Individual Subsystem” on page 6-4
Generate separate functions with no arguments, and optionally place the subsystem code in a separate file.	“Generate Subsystem Code as Separate Function and Files” on page 6-10
Generate a single reentrant function for a subsystem that is included in multiple places within a model.	“Generate Reusable Function for Identical Subsystems Within a Model” on page 6-11

To...	See...
Generate a single reentrant function for a subsystem that is included in multiple places in a model reference hierarchy.	“Generate Reusable Code for Subsystems Shared Across Models” on page 6-28 and “Code Reuse For Subsystems Shared Across Models” on page 6-20

Note: If you generate code for a virtual subsystem, code generator treats the subsystem as atomic and generates the code accordingly. The resulting code can change the execution behavior of your model, for example, by applying algebraic loops, and therefore introduce inconsistencies with the simulation behavior. Declare virtual subsystems as atomic subsystems, which makes simulation and execution behavior consistent for your model consistent.

Subsystem Code Dependence

Code generated from nonvirtual subsystems may or may not be completely independent of the generating model. When generating code for a subsystem, the code may reference global data structures of the model, even if the subsystem code is in a separate file. Each subsystem code file contains `include` directives and comments describing the dependencies. The code generator checks for cyclic file dependencies and warns about them at build time. For descriptions of how generated code is packaged, see “Generated Source Files and File Dependencies” on page 19-38.

To generate subsystem code that is independent of the generating model, place the subsystem in a library and configure it as a reusable subsystem. For more information, see “Code Reuse For Subsystems Shared Across Models” on page 6-20.

Generate Code and Executables for Individual Subsystem

You can generate code and build an executable for a subsystem within a model. The code generation and build process uses the code generation and build parameters of the root model.

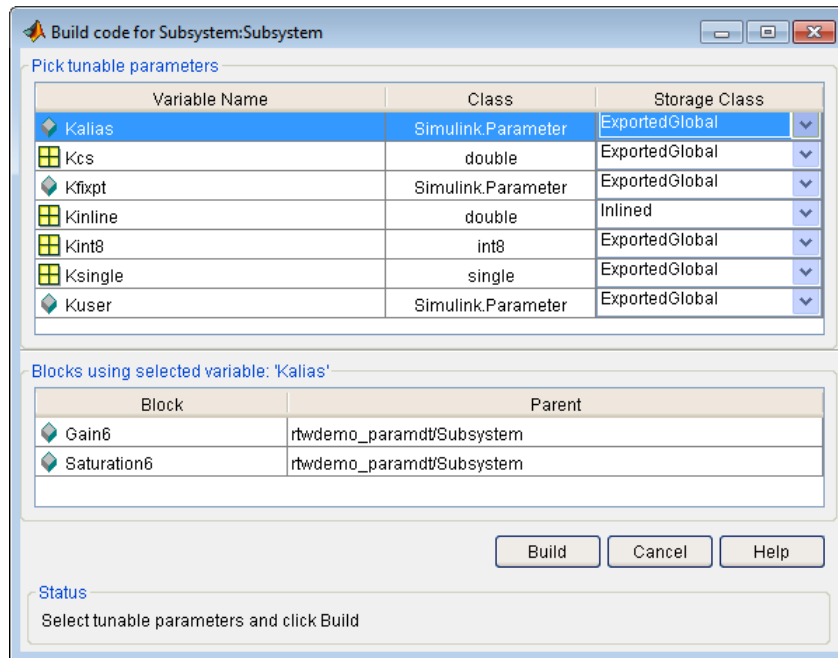
- 1 In the Configuration Parameters dialog box, set up the code generation and build parameters, similar to setting up the code generation for a model.
- 2 Right-click the Subsystem block. From the context menu, select **C/C++ Code > Build This Subsystem** from the context menu.

Alternatively, in the current model, click a subsystem and then from the **Code** menu, select **C/C++ Code > Build Selected Subsystem**.

Note When you select **Build This Subsystem**, if the model is operating in external mode, the build process automatically turns off external mode for the duration of the build. The code generator restores external mode upon completion of the build process.

- 3 The **Build code for Subsystem** window displays a list of the subsystem parameters. The upper pane displays the name, class, and storage class of each variable (or data object) that is referenced as a block parameter in the subsystem. When you select a parameter in the upper pane, the lower pane shows the blocks that reference the parameter and the parent system of each block.

The **Storage Class** column contains a menu for each row. The menu options set the storage class or inline the parameter. To declare a parameter to be tunable, set the **Storage Class** to a value other than **Inlined**.



For more information on tunable and inlined parameters and storage classes, see “Block Parameter Representation in the Generated Code” on page 14-43.

- 4 After selecting tunable parameters, **Build** to initiate the code generation and build process.
- 5 The build process displays status messages in the MATLAB Command Window. When the build is complete, the generated executable is in your working folder. The name of the generated executable is *subsystem.exe* (on PC platforms) or *subsystem* (on The Open Group UNIX platforms). *subsystem* is the name of the source subsystem block.

The generated code is in a build subfolder, named *subsystem_target_rtw*. *subsystem* is the name of the source subsystem block and *target* is the name of the target configuration.

When you generate code for a subsystem, you can generate an S-function by selecting **Code > C/C++ Code > Generate S-Function**, or you right-click the subsystem block and select **C/C++ Code > Build This Subsystem** from the context menu. For more

information on S-functions, see “Automate S-Function Generation with S-Function Builder” on page 24-22.

Subsystem Build Limitations

The following limitations apply to building subsystems:

- Subsystem build does not support a subsystem that has a function-call trigger input or a function-call output.
- When you right-click a subsystem block and select **C/C++ Code > Build This Subsystem** from the context menu to build a subsystem that includes an Outport block for which the **Data type** parameter specifies a bus object, you must address errors that result from setting signal labels. To configure the software to display these errors, in the Configuration Parameters dialog box for the parent model, on the **Diagnostics > Connectivity** pane, set the **Signal label mismatch** parameter to error.
- When a subsystem is in a triggered or function-call subsystem, the right-click build process might fail if the subsystem code is not sample-time independent. To find out whether a subsystem is sample-time independent:
 - 1 Copy all blocks in the subsystem to an empty model.
 - 2 In the Configuration Parameters dialog box, on the **Solver** pane, set:
 - a **Type** to **Fixed-step**.
 - b **Periodic sample time constraint** to **Ensure sample time independent**.
 - c Click **Apply**.
 - 3 Update the model. If the model is sample-time dependent, Simulink generates an error in the process of updating the diagram.

Inline Subsystem Code

You can configure a nonvirtual subsystem to inline the subsystem code with the model code. In the Subsystem Parameters dialog box, setting the **Function packaging** parameter to **Auto** or **Inline** inlines the generated code of the subsystem.

The **Auto** option is the default. When there is only one instance of a subsystem in the model, the **Auto** option inlines the subsystem code. When multiple instances of a subsystem exist, the **Auto** option results in a single copy of the function (as a reusable function). For function-call subsystems with multiple callers, the subsystem code is generated as if you specified **Nonreusable function**.

To inline subsystem code, select **Inline**. The **Inline** option explicitly directs the code generator to inline the subsystem unconditionally.

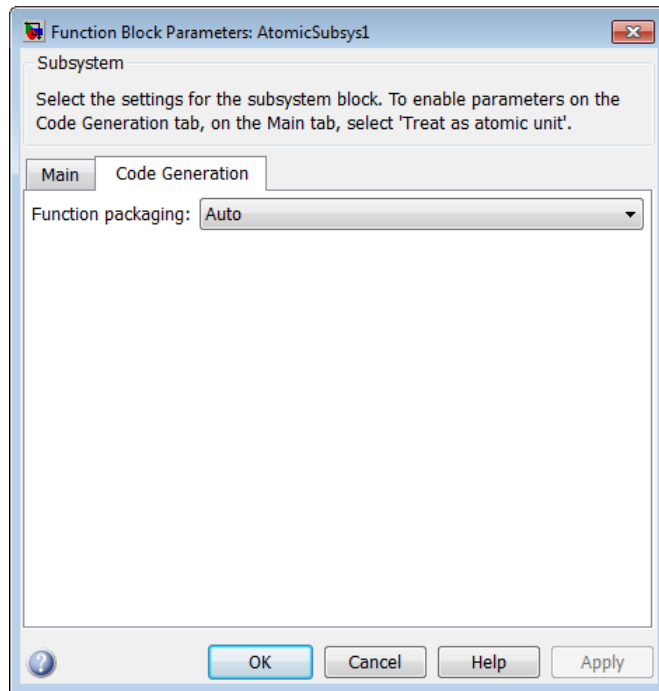
Configure Subsystem to Inline Code

To configure your subsystem for inlining:

- 1 Right-click the Subsystem block. From the context menu, select **Block Parameters (Subsystem)**.
- 2 In the Subsystem Parameters dialog box, if the subsystem is virtual, select **Treat as atomic unit**. This option makes the subsystem nonvirtual. On the **Code Generation** tab, the **Function packaging** option is now available.

If the system is already nonvirtual, the **Function packaging** option is already selected.

- 3 Click the **Code Generation** tab and select **Auto** or **Inline** from the **Function packaging** parameter.



- 4 Click **Apply** and close the dialog box.

The border of the subsystem thickens, indicating that it is nonvirtual.

When you generate code from your model, the code generator inlines subsystem code within *model.c* or *model.cpp* (or in its parent system's source file). You can identify this code by system/block identification tags, such as:

```
/* Atomic SubSystem Block: <Root>/AtomicSubsys1 */
```

Exceptions to Inlining

There are certain cases in which the code generator does not inline a nonvirtual subsystem, even though the **Inline** option is selected.

- If the subsystem is a function-call subsystem that is called by a noninlined S-function, the **Inline** option is ignored. Noninlined S-functions make calls by using function

pointers. Therefore, the function-call subsystem must generate a function with all arguments present.

- In a feedback loop involving function-call subsystems, the code generator forces one of the subsystems to be generated as a function instead of inlining it. Based on the order in which the subsystems are sorted internally, the software selects the subsystem to be generated as a function.
- If a subsystem is called from an S-function block that sets the option `SS_OPTION_FORCE_NONINLINED_FCNCALL` to `TRUE`, it is not inlined. When user-defined Async Interrupt blocks or Task Sync blocks are present, this result might occur. Such blocks must be generated as functions. These blocks are located in the `vxlib1` block library and use the `SS_OPTION_FORCE_NONINLINED_FCNCALL` option. This library demonstrates integration with an example RTOS (VxWorks).³

Note: You can use the blocks in the `vxlib1` on page 5-2 library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for your target environment.

3. VxWorks is a registered trademark of Wind River Systems, Inc.

Generate Subsystem Code as Separate Function and Files

To generate both a separate subsystem function and a separate file for a subsystem in a model:

- 1 Right-click a **Subsystem** block. From the context menu, select **Block Parameters (Subsystem)**.
- 2 In the Subsystem Parameters dialog box, if the subsystem is virtual, select **Treat as atomic unit**. On the **Code Generation** tab, the **Function packaging** parameter is now available.
- 3 Click the **Code Generation** tab and select **Nonreusable** function from the **Function packaging** parameter. The **Nonreusable** function option enables two parameters:
 - The “Function name options” parameter controls the naming of the generated function.
 - The “File name options” parameter controls the naming of the generated file.
- 4 Set the **Function name options** parameter.
- 5 Set the **File name options** parameter to a value other than **Auto**. If you are generating a reusable function for your subsystem, see “Generate Reusable Function for Identical Subsystems Within a Model” on page 6-11 or “Generate Reusable Code for Subsystems Shared Across Models” on page 6-28.
- 6 Click **Apply** and close the dialog box.

Generate Reusable Function for Identical Subsystems Within a Model

In the Subsystem Parameters dialog box, the **Function packaging** parameter option `Nonreusable function` generates functions that use global data. The `Reusable function` option generates reusable functions that have data passed as arguments (enabling them to be reentrant). Selecting `Reusable function` generates a function with arguments that allows the subsystem code to be shared by other instances of it in the model. This action supports less code instead of replicating the code for each instance of a subsystem or each time it is called.

To determine reusability of the subsystem code, the code generator performs a checksum to determine if subsystems are identical. The generated function has arguments, for example, for block inputs and outputs (`rtB_*`), continuous states (`rtDW_*`), parameters (`rtP_*`).

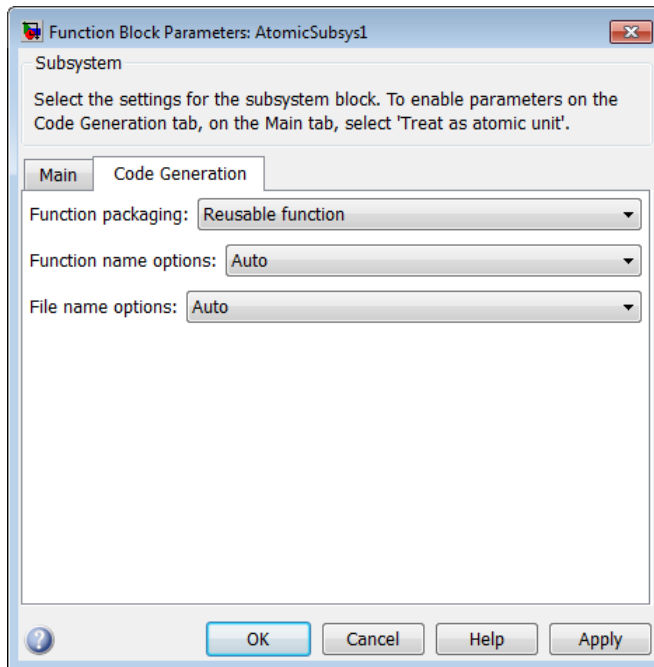
Note: In the generated code, the call interface is subject to change from release to release. Therefore, do not directly call reusable functions from external code.

To generate one reusable function for identical subsystems within a model:

- 1 Right-click the Subsystem block. From the context menu, select **Block Parameters (Subsystem)**.
- 2 In the Subsystem Parameters dialog box, if the subsystem is virtual, select **Treat as atomic unit**. On the **Code Generation** tab, the **Function packaging** menu is now available.

If the subsystem is already nonvirtual, the **Function packaging** menu is already selected.

- 3 Click the **Code Generation** tab and select `Reusable function` for the **Function packaging** parameter.



For more information about this setting, see “Considerations for Function Packaging Options Auto and Reusable function” on page 6-13.

- 4 Set the function name using the “Function name options” parameter.

Note: If you do not choose `Auto`, for other Subsystem blocks that you want to share this code, specify the same function name for those Subsystem blocks.

- 5 Set the file name using the “File name options” parameter to a value other than `Auto`. If your generated code is under source control, a value other than `Auto` prevents the generated file name from changing due to unrelated model modifications.

Note: For other Subsystem blocks that you want to share this code, specify the same file name for those Subsystem blocks.

- 6 Click **Apply** and close the dialog box.

For a summary of code reuse limitations, see “Code Reuse Limitations for Subsystems” on page 6-17.

Considerations for Function Packaging Options `Auto` and `Reusable function`

When you want multiple instances of a subsystem to be represented as one reusable function, you can designate each one of them as `Auto` or as `Reusable function`. Use one or the other, because using both creates two reusable functions, one for each specification. The outcomes of these choices differ only when reuse is not possible. Selecting `Auto` does not allow control of the function or file name for the subsystem code.

The `Reusable function` and `Auto` options both try to determine if multiple instances of a subsystem exist and if the code can be reused. When reuse is not possible, there are differences in the options behavior:

- `Auto` yields inlined code. If circumstances prohibit inlining, then the generated code is separate functions for each subsystem instance.
- `Reusable function` yields a separate function with arguments for each subsystem instance in the model.

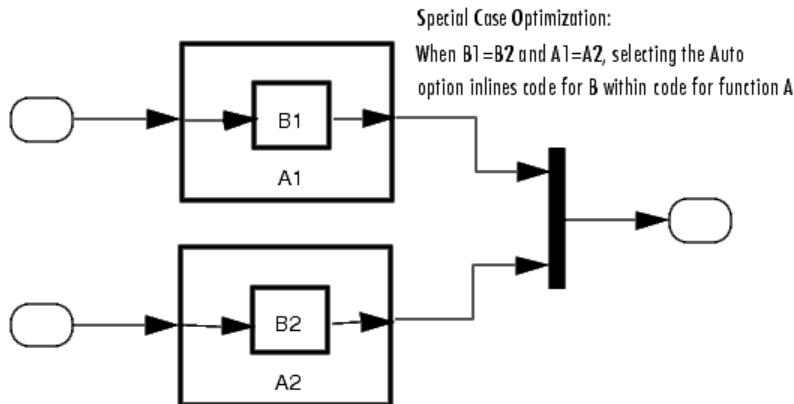
Code Reuse for Subsystems with Mask Parameters

The code generator can produce reusable (reentrant) code for a model containing identical atomic subsystems. Selecting the `Reusable function` option for **Function packaging** enables such code reuse, and causes a single function with arguments to be generated that is called when an identical atomic subsystem executes. See “Subsystems” for details and restrictions on the use of this option.

Mask parameters become arguments to reusable functions. However, for reuse to occur, each instance of a reusable subsystem must declare the same set of mask parameters. If, for example subsystem A has mask parameters `b` and `K`, and subsystem B has mask parameters `c` and `K`, then code reuse is not possible, and the code generator produces separate functions for A and B.

Optimize Code for Identical Nested Subsystems

The **Function packaging** parameter **Auto** option can optimize code in situations in which identical subsystems contain other identical subsystems, by both reusing and inlining generated code. Suppose a model, such as the one shown in Reuse of Identical Nested Subsystems, contains identical subsystems A1 and A2. A1 contains subsystem B1, and A2 contains subsystem B2, which are identical. In such cases, the **Auto** option causes one function to be generated which is called for both A1 and A2. This function contains one piece of inlined code to execute B1 and B2. This optimization generates less code which improves execution speed.



Reuse of Identical Nested Subsystems

Generate Reusable Code for Subsystems Containing S-Function Blocks

There are several requirements that need to be met in order for subsystems containing S-function blocks to be reused. For the list of requirements, see “S-Functions That Support Code Reuse” on page 24-152.

When you select the **Reusable** function option, two additional options are enabled, **Function name options** and **File name options**. If you use these fields to enter a function name and/or a file name, you must specify exactly the same function name and file name for each instance of identical subsystems for the code generator to reuse the subsystem code. For an example, follow the procedure in “Generate Reusable Function for Identical Subsystems Within a Model” on page 6-11.

Generate Reusable Code from Stateflow Charts

You can generate reusable code from a Stateflow chart, or from a subsystem containing a chart, *except* when the Stateflow chart contains exported graphical functions.

Code Reuse Limitations for Subsystems

The code generator uses a checksum to determine whether subsystems are identical and reusable. Subsystem code is not reused, if:

- In blocks and data objects, you use symbols to specify dimensions.
- A port used by multiple instances of a subsystem has different sample times, data types, complexity, frame status, or dimensions across the instances.
- The output of a subsystem is marked as a global signal.
- Subsystems contain identical blocks with different names or parameter settings.
- The output of a subsystem is connected to a Merge block, and the output of the Merge block is a custom storage class that is implemented in the C code as memory that is nonaddressable (for example, `BitField`).
- The input of a subsystem is nonscalar and has a custom storage class that is implemented in the C code as memory that is nonaddressable.
- A masked subsystem has a parameter that is nonscalar and has a custom storage class that is implemented in the C code as memory that is nonaddressable.
- A function-call subsystem uses mask parameters of any kind when you set the model configuration parameter “Default parameter behavior” to `Tunable`. To reuse the masked function-call subsystem, you can place the masked subsystem inside a new atomic subsystem without a mask, and move the Trigger block from the masked subsystem into the atomic subsystem.
- A block in the subsystems uses a *partially tunable expression*. Some partially tunable expressions can disable code reuse.

Partially tunable expressions are expressions that contain one or more tunable variables in addition to an expression that is not tunable. For example, suppose that you create the tunable variable `K` with value `15.23` and the tunable variable `P` with value `[5;7;9]`. The expression `K+P'` is a partially tunable expression because the expression `P'` is not tunable. For more information about tunable expression limitations, see “Tunable Expression Limitations” on page 14-49.

If you select `Reusable function`, and the code generator determines that you cannot reuse the code for a subsystem, it generates a separate function that is not reused. The code generation report might show that the separate function is reusable, even if only one subsystem uses it. If you prefer that subsystem code be inlined in such circumstances rather than deployed as functions, choose `Auto` for the **Function packaging** option.

Blocks That Prevent Code Reuse

Use of the following blocks in a subsystem can also prevent the subsystem code from being reused:

- Scope blocks (with data logging enabled)
- S-Function blocks that fail to meet certain criteria (see “S-Functions That Support Code Reuse” on page 24-152)
- To File blocks (with data logging enabled)
- To Workspace blocks (with data logging enabled)

Code Reuse Limitations for Subsystems Shared Across Referenced Models

The code generator uses a checksum to determine whether reusable library subsystems are identical. The code generator places the reusable library subsystem code in the shared utilities folder, and the reusable code is independent of the generated code of the top model or the referenced model. For example, the reusable library subsystem code does not include *model.h* or *model_types.h*.

Reusable code that is generated to the shared utilities folder and is dependent on the model code does not compile. If the code generator determines that the reusable library subsystem code is dependent on the model code, the reusable subsystem code is not generated to the shared utilities folder. The following cases can generate code that is dependent on the model code, when the reusable library subsystem:

- Contains a block that uses time-related functionality, such as a Step block, or continuous time or multirate blocks.
- Contains one or more Model blocks.
- Contains subsystems that are not inlined or a reusable library subsystem.
- Contains a signal that is not an **Auto** storage class. Variables of non-**Auto** storage classes are generated to *model.h*.
- Contains a parameter that is not an **Auto** storage class.
- Contains a user-defined type where **Data Scope** is not set to **Exported**. The code generator might place the type definition in *model_types.h*.
- Is a variant subsystem that generates preprocessor conditionals. Preprocessor directives defining the variant objects are included in *model_types.h*.

Related Examples

- “Determine Why Subsystem Code Is Not Reused” on page 6-36

Code Reuse For Subsystems Shared Across Models

To reuse common functionality, you can include multiple instances of a subsystem:

- Within a single model, which is a top model or part of model reference hierarchy
- Across multiple referenced models in a model reference hierarchy
- Across multiple top models that contain Model blocks
- Across multiple top models that do not include Model blocks

To generate a reusable function for a subsystem which is included in multiple models:

- If the subsystem is in a model reference hierarchy, set the configuration parameter, “Shared code placement” to `Auto`. Otherwise, for each model that uses the subsystem, set **Shared code placement** to `Shared location`. The **Shared code placement** parameter is in the Configuration Parameters dialog box, on the **Code Generation > Interface** pane.
- The subsystem must be defined in a library and configured for reuse. This subsystem is referred to as a *reusable library subsystem*. For more information, see “Reusable Library Subsystem” on page 6-21.

For an example, see “Generate Reusable Code for Subsystems Shared Across Models” on page 6-28.

The code generator performs a checksum to determine reusability. There are cases when the code generator cannot reuse subsystem code. For more information, see “Code Reuse Limitations for Subsystems” on page 6-17.

Related Examples

- “Determine Why Subsystem Code Is Not Reused” on page 6-36

Reusable Library Subsystem

A reusable library subsystem is a subsystem included in a library that is configured for reuse. The Subsystem parameters must be set as follows:

- **Treat as an atomic unit** is selected.
- On the **Code Generation** tab:
 - **Function packaging** is set to `Reusable` function.
 - **Function name options**
and **File name options** are set to `Auto` or `Use subsystem name`.

For more information on creating a library, see “Libraries”. For an example of creating a reusable library subsystem, see “Generate Reusable Code for Subsystems Shared Across Models” on page 6-28.

Code Generation of a Reusable Library Subsystem

For incremental code generation, if the reusable library subsystem changes, a rebuild of itself and its parents occurs. During the build, if a matching function is not found, a new instance of the reusable function is generated into the shared utilities folder. If a different matching function is found from previous builds, that function is used, and a new reusable function is not emitted.

For subsequent builds, unused files are not replaced or deleted from your folder. During development of a model, when many obsolete shared functions exist in the shared utilities folder, you can delete the folder and regenerate the code. If all instances of a reusable library subsystem are removed from a model reference hierarchy and you regenerate the code, the obsolete shared functions remain in the shared utilities folder until you delete them.

If a model changes such that the change might cause different generated code for the subsystem, a new reusable function is generated. For example, model configuration parameters that modify code comments might cause different generated code for the subsystem even if the reusable library subsystem did not change.

Reusable Library Subsystem Code Placement and Naming

The code generator uses checksums to determine reusability. The generated code of a reusable library subsystem is independent of the generated code of the model. Code for the reusable library subsystem is generated to the shared utility folder, `slprj/target/_sharedutils`, instead of the model reference hierarchy folders. The generated code for the supporting types, which are generated to the `.h` file, are also in the shared utilities folder.

In the Subsystem Parameters dialog box, the **Function name options** and **File name options** must be set to **Auto** or **Use subsystem name**. For unique naming, reusable function names have a checksum appended to the reusable library subsystem name. For example, the code and files for a subsystem, `SS1`, which links to a reusable library subsystem, `RLS`, might be:

- Function name: `RLS_mgdj1ngd`
- File name: `RLS_mgdj1nd.c` and `RLS_mgdj1nd.h`

Reusable Library Subsystem in the Top Model

In a model reference hierarchy, if an instance of the reusable library subsystem is in the top model, then on the **Model Referencing** pane of the Configuration Parameters dialog box, you must select the **Pass fixed-size scalar root input by value for code generation** parameter. If you do not select the parameter, a separate shared function is generated for the reusable library subsystem instance in the top model, and a reusable function is generated for instances in the referenced models.

Reusable Library Subsystem Connected to Root Output

If a reusable library subsystem is connected to the root output, reuse does not happen with identical subsystems that are not connected to the root output. However, you can set **Pass reusable system outputs as to Individual arguments** on the **Optimizations > Signals and Parameters** pane to make sure that reuse occurs between these subsystems. This parameter requires an Embedded Coder license.

Code Generation of Constant Parameters

The code generator attempts to generate constant parameters to the shared utilities folder first. If constant parameters are not generated to the shared utilities folder, they are defined in the top model in a global constant parameter structure. The declaration of the structure, `ConstParam_model`, is in `model.h`:

```
/* Constant parameters (auto storage) */
typedef struct {
    /* Expression: [1 2 3 4 5 6 7]
     * Referenced by: '<Root>/Constant'
     */
    real_T Constant_Value[7];

    /* Expression: [7 6 5 4 3 2 1]
     * Referenced by: '<Root>/Gain'
     */
    real_T Gain_Gain[7];
} ConstParam_model;
```

The definition of the constant parameters, `model_constP`, is in:

```
/* Constant parameters (auto storage) */
const ConstParam_model model_ConstP = {
    /* Expression: [1 2 3 4 5 6 7]
     * Referenced by: '<Root>/Constant'
     */
    { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 },

    /* Expression: [7 6 5 4 3 2 1]
     * Referenced by: '<Root>/Gain'
     */
    { 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0 }
};
```

The `model_constP` is passed as an argument to referenced models. For more information on how shared constants are generated, see “Shared Constant Parameters for Code Reuse” on page 6-24.

Shared Constant Parameters for Code Reuse

You can share the generated code for constant parameters across models if:

- Constant parameters are shared in a model reference hierarchy, or
- On the **Code Generation > Interface** pane, the model configuration parameter “Shared code placement” is set to **Shared location**.

If you do not want to generate shared constants, and **Shared code placement** is set to **Shared location**, set the parameter `GenerateSharedConstants` to `off`. For example, to turn off shared constants for the current model, in the Command Window, type the following.

```
set_param(gcs, 'GenerateSharedConstants', 'off');
```

The shared constant parameters are generated individually to the `const_params.c` file, which is placed in the shared utilities folder `slprj/target/_sharedutils`.

For example, if a constant has multiple uses within a model reference hierarchy where the top model is named `topmod`, the code for the shared constant is as follows:

- In the shared utility folder, `slprj/grt/_sharedutils`, the constant parameters are defined in `const_params.c` and named `rtCP_pooled_` appended to a unique checksum:

```
extern const real_T rtCP_pooled_lfcjjmohiecj[7];
const real_T rtCP_pooled_lfcjjmohiecj[7] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };

extern const real_T rtCP_pooled_ppphohdbfcba[7];
const real_T rtCP_pooled_ppphohdbfcba[7] = { 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0 };
```

- In `top_model_private.h` or in a referenced model, `ref_model_private.h`, for better readability, the constants are renamed as follows:

```
extern const real_T rtCP_pooled_lfcjjmohiecj[7];
extern const real_T rtCP_pooled_ppphohdbfcba[7];

#define rtCP_Constant_Value    rtCP_pooled_lfcjjmohiecj /* Expression: [1 2 3 4 5 6 7]
                               * Referenced by: '<Root>/Constant' */
#define rtCP_Gain_Gain        rtCP_pooled_ppphohdbfcba /* Expression: [7 6 5 4 3 2 1]
                               * Referenced by: '<Root>/Gain' */
```

- In `topmod.c` or `refmod.c`, the call site might be:

```
for (i = 0; i < 7; i++) {
    topmod_Y.Out1[i] = (topmod_U.In1 + rtCP_Constant_Value[i]) * rtCP_Gain_Gain[i];
}
```

The code generator attempts to generate all constants as individual constants to the `const_params.c` file in the shared utilities folder. Otherwise, constants are generated as described in “Code Generation of Constant Parameters” on page 6-23.

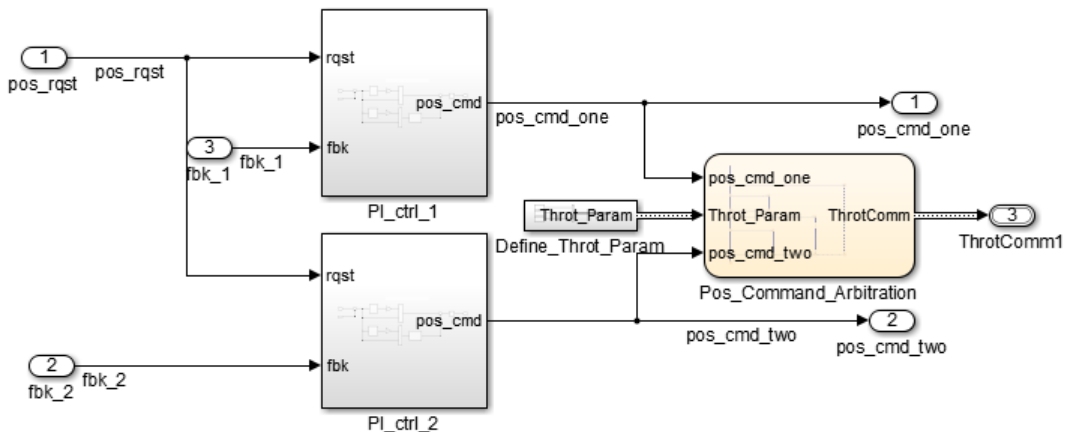
Suppress Shared Constants in the Generated Code

You can choose whether or not the code generator produces shared constants and shared functions. You may want to be able to keep the code and data separate between subsystems, or you may find that sharing constants results in a memory shortage during code generation.

You can change this parameter programmatically using the parameter `GenerateSharedConstants` with `set_param` and `get_param`.

In the following example, when `GenerateSharedConstants` is set to `on`, the code generator defines the constant values in the `_sharedutils` folder in the `const_params.c` file. When `GenerateSharedConstants` is set to `off`, the code generator defines the constant values in a nonshared area, in the `model_ert_rtw` file in the `model_data.c` file.

Open the model `rtwdemo_throttlecntrl`:



In the Configuration parameters dialog box, on the **Code Generation > Interface** pane, verify that “Shared code placement” is set to `Shared location`. If `Shared`

code placement is set to `Auto`, the `GenerateSharedConstants` setting is ignored. If you try to set the parameter value, an error message appears. The default value of `GenerateSharedConstants` is `on`.

In the Command Window, set `GenerateSharedConstants` to `on`:

```
>> set_param('rtwdemo_throttlecntrl','GenerateSharedConstants','on')
```

You see the shared constant definitions in the folder `slprj/grt/_sharedutils`, in the file `const_params.c`:

```
extern const real_T rtCP_pooled_H4eTKtECwveN[9];
const real_T rtCP_pooled_H4eTKtECwveN[9] = { 1.0, 0.75, 0.6, 0.0, 0.0, 0.0, 0.6,
    0.75, 1.0 };

extern const real_T rtCP_pooled_SghuHxKVKGHD[9];
const real_T rtCP_pooled_SghuHxKVKGHD[9] = { -1.0, -0.5, -0.25, -0.05, 0.0, 0.05,
    0.25, 0.5, 1.0 };

extern const real_T rtCP_pooled_WqWb2t17NA2R[7];
const real_T rtCP_pooled_WqWb2t17NA2R[7] = { -1.0, -0.25, -0.01, 0.0, 0.01, 0.25,
    1.0 };

extern const real_T rtCP_pooled_Ygna10wM3c14[7];
const real_T rtCP_pooled_Ygna10wM3c14[7] = { 1.0, 0.25, 0.0, 0.0, 0.0, 0.25, 1.0
};
```

In the Command Window, set `GenerateSharedConstants` to `off`:

```
>> set_param('rtwdemo_throttlecntrl','GenerateSharedConstants','off')
```

You can see the unshared constants in the folder `rtwdemo_throttlecntrl_grt_rtw`, in the file `rtwdemo_throttlecntrl_data.c`:

```
/* Constant parameters (auto storage) */
const ConstP_rtwdemo_throttlecntrl_T rtwdemo_throttlecntrl_ConstP = {
    /* Pooled Parameter (Expression: P_OutMap)
    * Referenced by:
    *   '<S2>/Proportional Gain Shape'
    *   '<S3>/Proportional Gain Shape'
    */
    { 1.0, 0.25, 0.0, 0.0, 0.0, 0.25, 1.0 },

    /* Pooled Parameter (Expression: P_InErrMap)
    * Referenced by:
    *   '<S2>/Proportional Gain Shape'
    *   '<S3>/Proportional Gain Shape'
    */
    { -1.0, -0.25, -0.01, 0.0, 0.01, 0.25, 1.0 },

    /* Pooled Parameter (Expression: I_OutMap)
    * Referenced by:
```



```

* '<S2>/Integral Gain Shape'
* '<S3>/Integral Gain Shape'
*/
{ 1.0, 0.75, 0.6, 0.0, 0.0, 0.0, 0.6, 0.75, 1.0 },

/* Pooled Parameter (Expression: I_InErrMap)
* Referenced by:
* '<S2>/Integral Gain Shape'
* '<S3>/Integral Gain Shape'
*/
{ -1.0, -0.5, -0.25, -0.05, 0.0, 0.05, 0.25, 0.5, 1.0 }
};

```

Shared Constant Parameters Limitations

No shared constants or shared functions are generated for a model when:

- The model has a Code Replacement Library (CRL) that is specified for data alignment.
- The model is specified to replace data type names in the generated code.
- The **Memory Section** for constants is `MemVolatile` or `MemConstVolatile`.
- The parameter `GenerateSharedConstants` is set to `off`.

Individual constants are not shared, if:

- A constant is referenced by a non-inlined S-function.
- A constant has a user-defined type where **Data Scope** is not set to `Exported`.

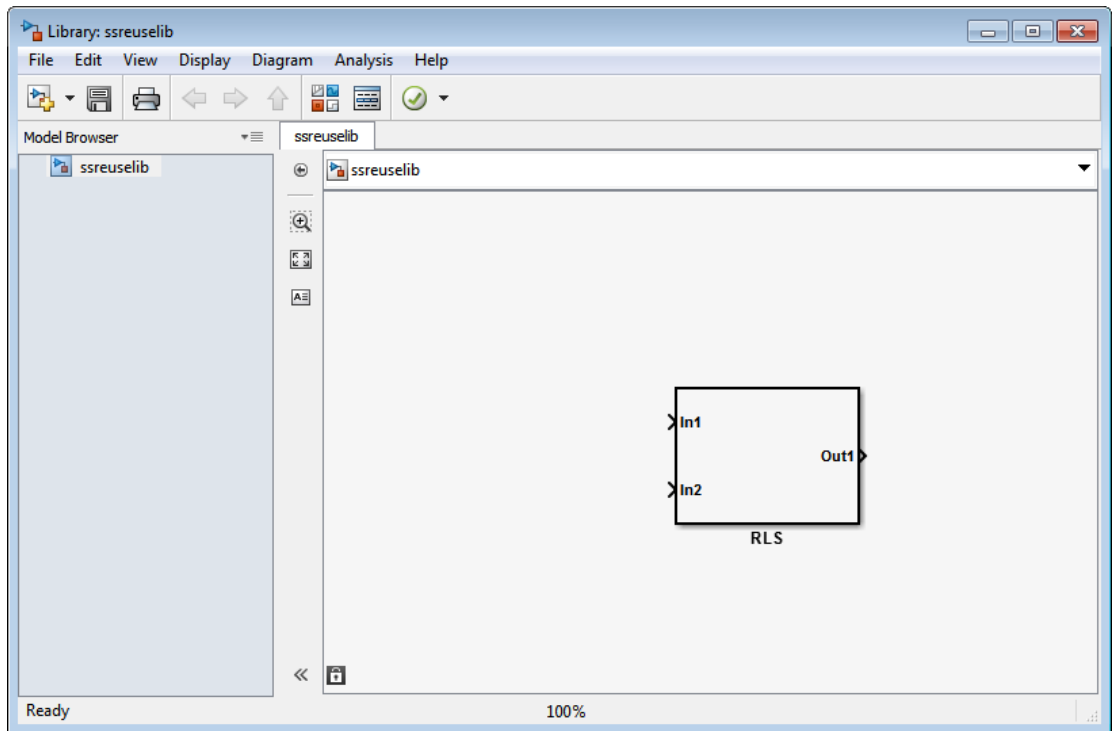
Generate Reusable Code for Subsystems Shared Across Models

This example shows how to configure a reusable library subsystem and generate a reusable function for a subsystem shared across referenced models. The result is reusable code for the subsystem, which is generated to the shared utility folder (`slprj/target/_sharedutils`).

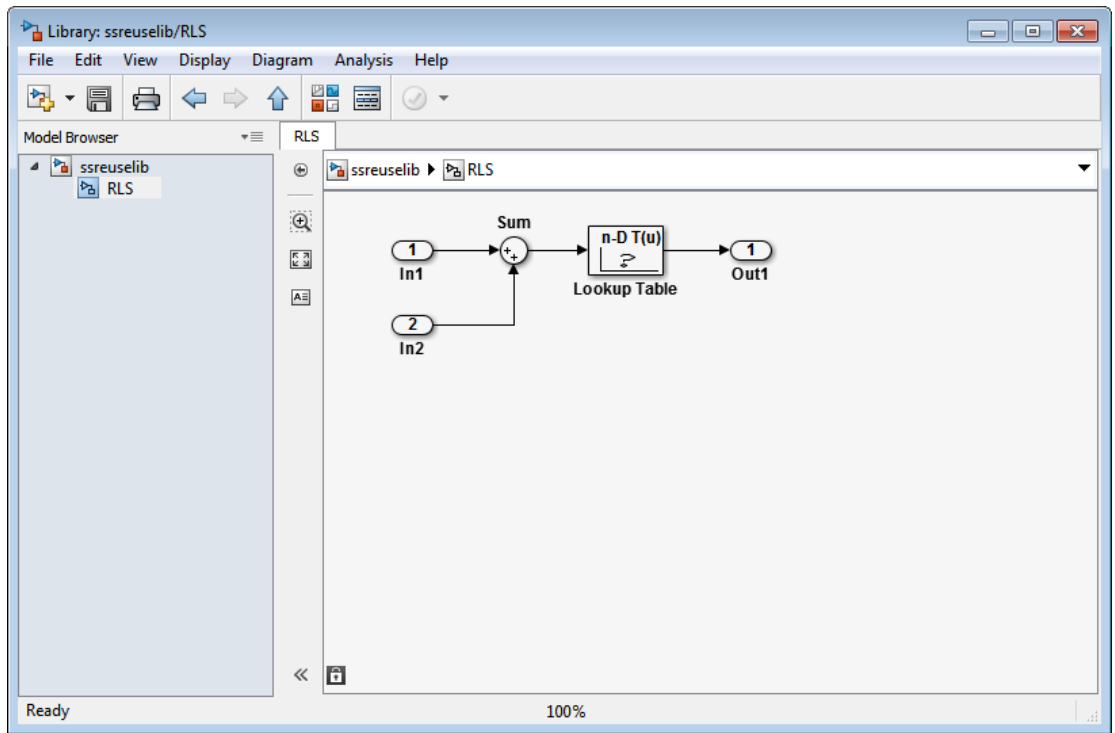
In this section...
“Create a reusable library subsystem.” on page 6-28
“Create the example model.” on page 6-31
“Set configuration parameters of the top model.” on page 6-33
“Create and propagate a configuration reference.” on page 6-33
“Generate and view the code.” on page 6-34

Create a reusable library subsystem.

- 1 In the Simulink Editor, select **File > New > Library**. Open `rtwdemo_ssreuse` to copy and paste subsystem `SS1` into the Library Editor. This action loads the variables for `SS1` into the base workspace. Rename the subsystem block to `RLS`.



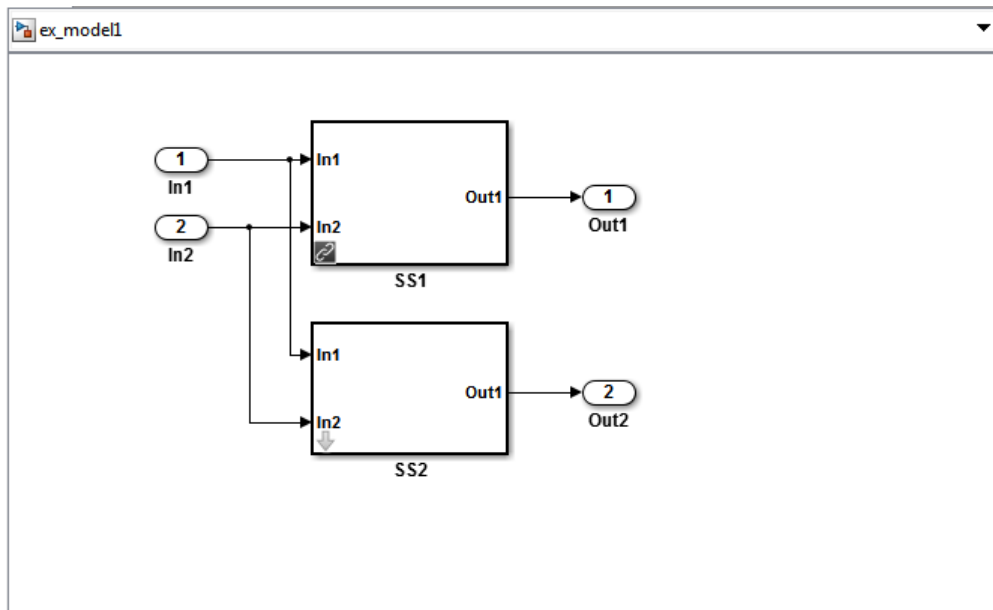
- 2 Click the Subsystem block and press **Ctrl+U** to view the contents of subsystem RLS.



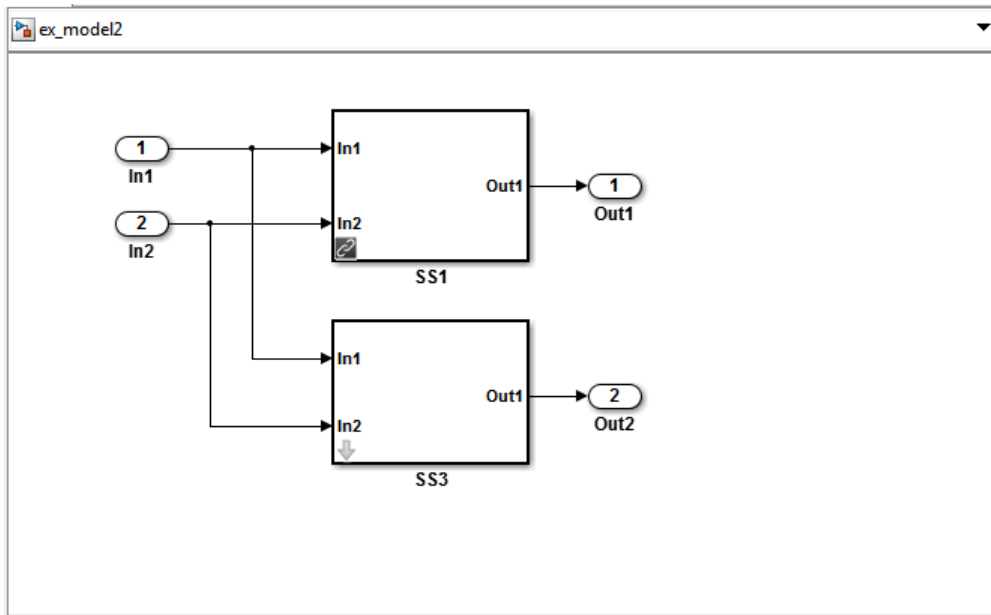
- 3 To configure the subsystem, in the Library editor, right-click RLS. In the context menu, select **Block Parameters(Subsystem)**. In the Subsystem Parameters dialog box, choose the following options:
 - Select **Treat as an atomic unit**.
 - On the **Code Generation** tab:
 - Set **Function packaging** to Reusable function.
 - Set **Function name options** and **File name options** to Auto.
- 4 Click **Apply** and **OK**.
- 5 Save the reusable library subsystem as `ssreuselib`, which creates a file, `ssreuselib.slx`.

Create the example model.

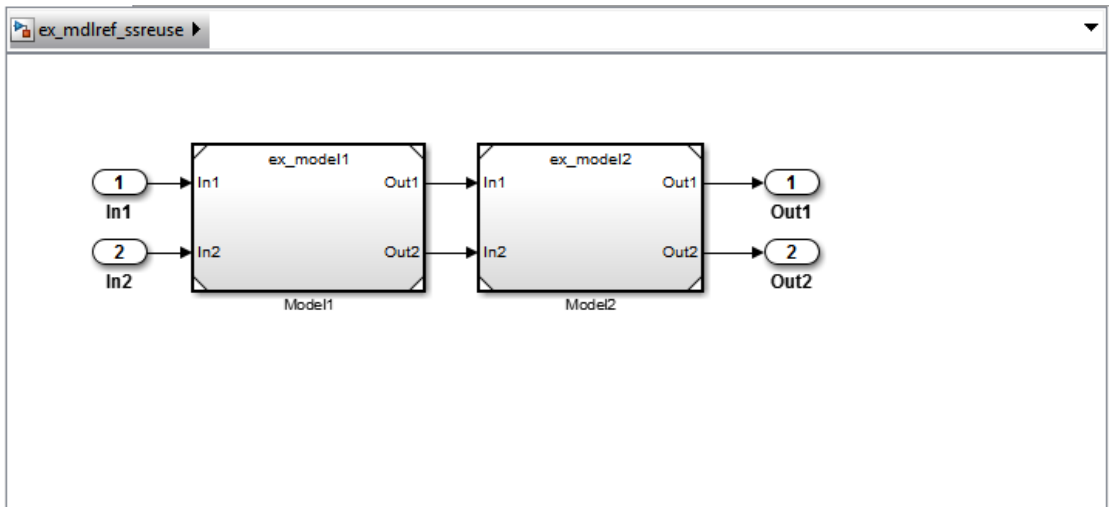
- 1 Create a model which includes one instance of RLS from `ssreuselib`. Name this subsystem `SS1`. Add another subsystem and name it `SS2`. Name the model `ex_model1`.



- 2 Create another model which includes one instance of RLS from `ssreuselib`. Name this subsystem `SS1`. Add another subsystem and name it `SS3`. Name the model `ex_model2`.



- 3 Create a top model with two model blocks that reference `ex_model1` and `ex_model2`. Save the top model as `ex_md1ref_ssreuse`.



Set configuration parameters of the top model.

- 1 With model `ex_mdhref_ssreuse` open in the Simulink Editor, select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box.
- 2 On the **Solver** pane, specify the **Type** as `Fixed-step`.
- 3 On the **Optimization > Signals and Parameters** pane, set “Pass reusable subsystem outputs as” to `Individual arguments`.
- 4 On the **Model Referencing** pane, select **Pass fixed-size scalar root inputs by value for code generation**.
- 5 On the **Code Generation > Report** pane, select **Create code generation report and Open report automatically**.
- 6 On the **Code Generation > Interface** pane, set the “Shared code placement” to `Shared location`.
- 7 On the **Code Generation > Symbols** pane, set the **Maximum identifier length** to `256`. This step is optional.
- 8 Click **Apply** and **OK**.

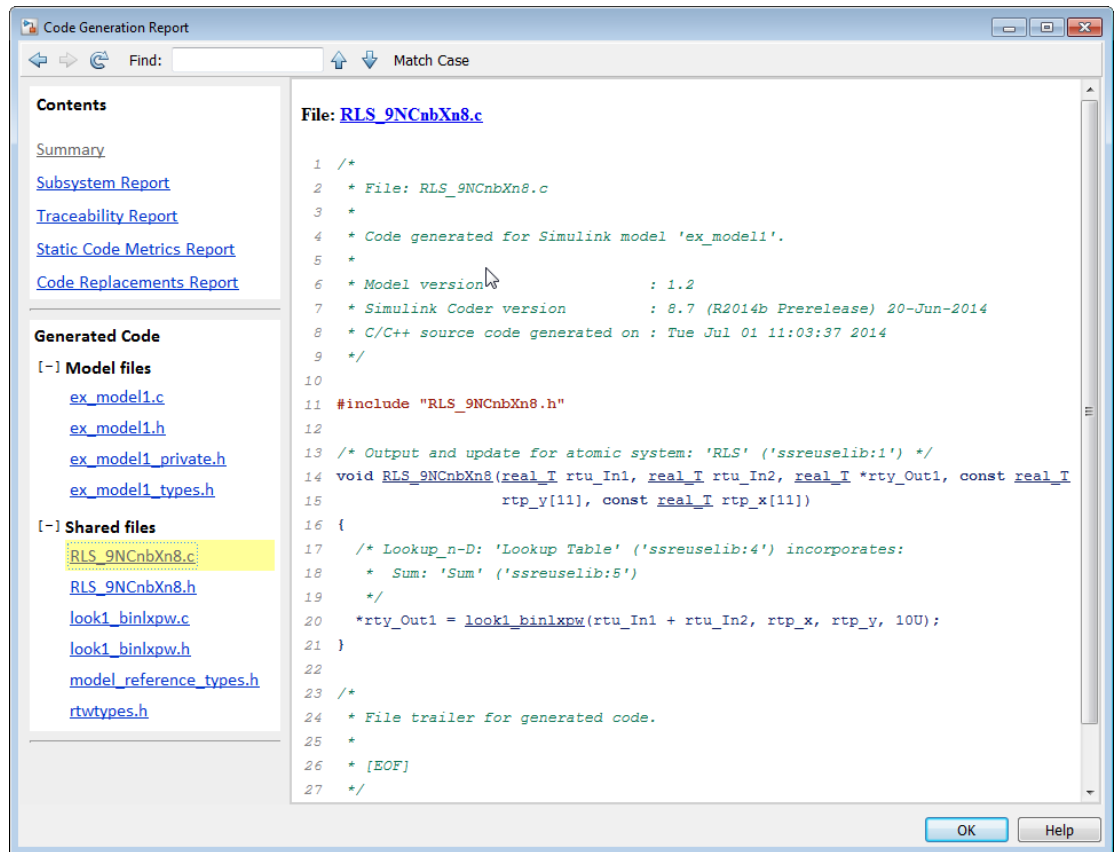
Create and propagate a configuration reference.

- 1 In the Simulink Editor, select **View > Model Explorer** to open the Model Explorer. In the left navigation column of the Model Explorer, expand the `ex_mdhref_ssreuse` node.
- 2 Right-click **Configuration** and select **Convert to Configuration Reference**.
- 3 In the Convert Active Configuration to Reference dialog box, click **OK**. This action converts the model configuration set to a configuration reference, `Simulink.ConfigSetRef`, and creates the configuration reference object, `configSetObj`, in the base workspace.
- 4 In the left navigation column, right-click **Reference (Active)** and select **Propagate to Referenced Models**.
- 5 In the Configuration Reference Propagation to Referenced Models dialog box, select the referenced models in the list. Click **Propagate**.

Now, the top model and referenced models use the same configuration reference, **Reference (Active)**, which points to a model configuration reference object, `configSetObj`, in the base workspace. When you save your model, you also need to export the `configSetObj` to a MAT-file.

Generate and view the code.

- 1 To generate code, in the Simulink Editor, press **Ctrl-B**. After the code is generated, the code generation report opens.
- 2 To view the code generation report for a referenced model, in the left navigation pane, in the **Referenced Models** section, select `ex_model1`. The code generation report displays the generated files for `ex_model1`.
- 3 In the left navigation pane, expand the **Shared files**. The code generator uses the reusable library subsystem name and a unique identifier to name the reused function. The code for subsystem SS1 is in `RLS_9NCnbXn8.c` and `RLS_9NCnbXn8.h`.



- 4 Click **Back** and navigate to the `ex_model2` code generation report. `ex_model2` uses the same source code, `RLS_9NCnbXn8.c` and `RLS_9NCnbXn8.h`, as the code for `ex_model1`. Your subsystem function and file names will be different.

Related Examples

- “Determine Why Subsystem Code Is Not Reused” on page 6-36
- “Generate Reusable Function for Identical Subsystems Within a Model” on page 6-11

More About

- “Code Generation of Subsystems” on page 6-2
- “Code Reuse For Subsystems Shared Across Models” on page 6-20
- “Code Reuse Limitations for Subsystems” on page 6-17
- “Libraries”

Determine Why Subsystem Code Is Not Reused

Due to the limitations described in “Code Reuse Limitations for Subsystems” on page 6-17, the code generator might not reuse generated code as you expect. To determine why code generated for a subsystem is not reused, see “Review Subsystems Section of HTML Code Generation Report” on page 6-36. If you cannot determine why based on the report, see “Compare Subsystem Checksum Data” on page 6-36.

Review Subsystems Section of HTML Code Generation Report

If you determine that the code generator does not generate code for a subsystem as reusable code, and you specified the subsystem as reusable, examine the Subsystems section of the HTML code generation report (see “Generate a Code Generation Report” on page 20-5). The Subsystems section contains:

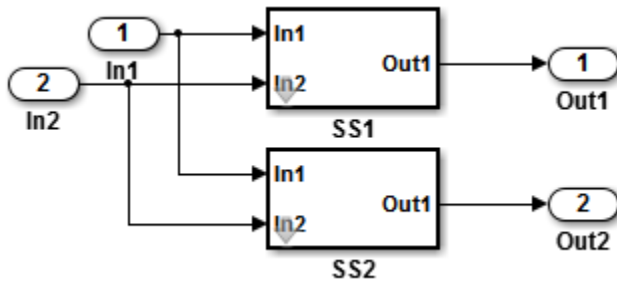
- A table that summarizes how nonvirtual subsystems were converted to generated code.
- Diagnostic information that describes why the contents of some subsystems were not generated as reusable code.

The Subsystems section also indicates the mapping of each noninlined subsystem in the model to functions or reused functions in the generated code. For an example, open and build the `rtwdemo_atomic` model.

Compare Subsystem Checksum Data

You can determine why subsystem code is not reused by accessing and comparing subsystem checksum data. The code generator determines whether subsystems are identical by comparing subsystem checksums, as noted in “Code Reuse Limitations for Subsystems” on page 6-17. For subsystem reuse across referenced models, this procedure might not catch every difference.

Consider the model, `rtwdemo_ssreuse`. `SS1` and `SS2` are instances of the same subsystem. In both instances the subsystem parameter **Function packaging** is set to `Reusable function`.



Use the method, `Simulink.SubSystem.getChecksum`, to get the checksum for a subsystem. Compare the results to determine why code is not reused.

- 1 Open the model `rtwdemo_ssreuse`. Save a copy of the model in a folder where you have write access.
- 2 In the model window, select subsystem `SS1`. In the command window, enter


```
SS1 = gcb;
```
- 3 In the model window, select subsystem `SS2`. In the command window, enter


```
SS2 = gcb;
```
- 4 Use the method, `Simulink.SubSystem.getChecksum`, to get the checksum for each subsystem. This method returns two output values: the checksum value and details on the input used to compute the checksum.

```
[chksum1, chksum1_details] = ...
Simulink.SubSystem.getChecksum(SS1);
[chksum2, chksum2_details] = ...
Simulink.SubSystem.getChecksum(SS2);
```

- 5 Compare the two checksum values. They should be equal based on the subsystem configurations.

```
isequal(chksum1, chksum2)
ans =
    1
```

- 6 To see how you can use `Simulink.SubSystem.getChecksum` to determine why the checksums of two subsystems differ, change the data type mode of the output port of `SS1` so that it differs from that of `SS2`.

- a Look under the mask of SS1. Right-click the subsystem. In the context menu, select **Mask > Look Under Mask**.
 - b In the block diagram of the subsystem, double-click the Lookup Table block to open the Subsystem Parameters dialog box.
 - c Click **Data Types**.
 - d Select **Saturate on integer overflow** and click **OK**.
- 7 Get the checksum for SS1. Compare the checksums for the two subsystems. This time, the checksums are not equal.

```
[chksum1, chksum1_details] = ...
Simulink.SubSystem.getChecksum(SS1);
isequal(chksum1, chksum2)
ans =
    0
```

- 8 After you determine that the checksums are different, find out why. The Simulink engine uses information, such as signal data types, some block parameter values, and block connectivity information, to compute the checksums. To determine why checksums are different, you compare the data used to compute the checksum values. You can get this information from the second value returned by `Simulink.SubSystem.getChecksum`, which is a structure array with four fields.

Look at the structure `chksum1_details`.

```
chksum1_details
```

```
chksum1_details =
    ContentsChecksum: [1x1 struct]
    InterfaceChecksum: [1x1 struct]
    ContentsChecksumItems: [287x1 struct]
    InterfaceChecksumItems: [53x1 struct]
```

`ContentsChecksum` and `InterfaceChecksum` are component checksums of the subsystem checksum. The remaining two fields, `ContentsChecksumItems` and `InterfaceChecksumItems`, contain the checksum details.

- 9 Determine whether a difference exists in the subsystem contents, interface, or both. For example:

```
isequal(chksum1_details.ContentsChecksum.Value,...
        chksum2_details.ContentsChecksum.Value)
ans =
```

```

    0
isequal(chksum1_details.InterfaceChecksum.Value,...
        chksum2_details.InterfaceChecksum.Value)
ans =
    1

```

In this case, differences exist in the contents.

- 10** Write a script like the following to find the differences.

```

idxForCDiffs=[];
for idx = 1:length(chksum1_details.ContentsChecksumItems)
    if (~strcmp(chksum1_details.ContentsChecksumItems(idx).Identifier, ...
               chksum2_details.ContentsChecksumItems(idx).Identifier))
        disp(['Identifiers different for contents item ', num2str(idx)]);
        idxForCDiffs=[idxForCDiffs, idx];
    end
    if (ischar(chksum1_details.ContentsChecksumItems(idx).Value))
        if (~strcmp(chksum1_details.ContentsChecksumItems(idx).Value, ...
                   chksum2_details.ContentsChecksumItems(idx).Value))
            disp(['Character vector values different for contents item ', num2str(idx)]);
            idxForCDiffs=[idxForCDiffs, idx];
        end
    end
    if (isnumeric(chksum1_details.ContentsChecksumItems(idx).Value))
        if (chksum1_details.ContentsChecksumItems(idx).Value ~= ...
           chksum2_details.ContentsChecksumItems(idx).Value)
            disp(['Numeric values different for contents item ', num2str(idx)]);
            idxForCDiffs=[idxForCDiffs, idx];
        end
    end
end
end

```

- 11** Run the script. The following example assumes that you named the script `check_details`.

```

check_details
Character vector values different for contents item 202

```

The results indicate that differences exist for index item 202 in the subsystem contents.

- 12** Use the returned index values to get the handle, identifier, and value details for each difference found.

```

chksum1_details.ContentsChecksumItems(202)

ans =

    Handle: 'rtwdemo_ssreuse/SS1/Lookup Table'
  Identifier: 'SaturateOnIntegerOverflow'
    Value: 'on'

```

The details identify the Lookup Table block parameter **Saturate on integer overflow** as the focus for debugging a subsystem reuse issue.

Code Generation of Functions and Function Callers in Simulink Coder

Modeling Functions and Callers for Code Generation

In this section...

“Functions and Callers” on page 7-2

“Input and Output Arguments” on page 7-2

“Function and Function Caller Definitions Across Models” on page 7-3

“Code Generation Files” on page 7-3

Functions and Callers

Use a Simulink Function block and a Function Caller block to instruct the code generator to generate C functions and function calls in the generated code for encapsulation and portability. A Simulink Function block is a nonreusable subsystem.

With a Simulink Function block and a Function Caller block, you can:

- Use nested function calls to call a function from a function.
- Choose to separate function definitions and calls into different models.
- Specify SIL and PIL simulations.
- Integrate code for multiple top models where the Simulink Function block and Function callers are in different models.
- Use global data to communicate between a server and its parent model. This data uses custom storage classes to customize how the data is communicated.
- Model a client and server application using the export functions modeling style.

Simulink Function blocks and Function Caller blocks do not honor the `MaxStackSize` parameter.

For more information, see “Simulink Functions”, “Diagnostics Using a Client-Server Architecture”, and Simulink Function block.

Input and Output Arguments

When you set up your model that contains Simulink Function blocks for code generation:

- Do not define the signals entering and leaving Argument Inport blocks and Argument Outport blocks in the Simulink Function definition with a storage class.

- Do not specify Argument Inport and Argument Outport blocks as test points.
- If you specify the data type of signals entering and leaving Argument Inport and Argument Outport blocks as a `Simulink.IntEnumType`, `Simulink.AliasType` or `Simulink.Bus` type, then you must specify the arguments as `Imported` or `Exported`, not `Auto`.
- The Simulink Function block and the Function Caller blocks must agree in data type, complexity, dimension, and number of arguments.

For more information, see `Argument Inport` and `Argument Outport`.

Function and Function Caller Definitions Across Models

You can define a Simulink Function block and Function Caller block in different models. When the code generator generates code for a model hierarchy, it can encounter either a Simulink Function block or a Function Caller block first. If the code generator finds the Simulink Function block first, the software uses the function definition from the Simulink Function block. If the code generator then encounters a Function Caller block that does not match the function definition, the code generator issues an error. This error prompts you to either change the Function Caller block to match the Simulink Function block or remove the `slprj` folder. Verify that the arguments and data types in the Function Caller blocks match the arguments and data types in the Simulink Function block. Regenerate code for the models involved.

If the code generator encounters a Function Caller block first, then the code generator uses the function definition derived from the Function Caller block. If the code generator then encounters a Simulink Function block with different arguments and data types from the Function Caller block, the code generator issues a warning message. Verify that the Function Caller blocks match the Simulink Function block. Regenerate the code where you have made changes.

Specifying two Simulink Function blocks with the same name is an error. Modify one of the blocks and remove the `slprj` folder.

Code Generation Files

In the build folders, the code generator creates different files depending on the setting you choose.

When one of the following is true, the code generator creates files as shown in the table.

- The system target file is `grt.tlc`.
- The system target file is `ert.tlc` and you do not have an Embedded Coder license.
- The system target file is `ert.tlc` and you have an Embedded Coder license. For the **Code Generation > Code Placement > File packaging format** parameter, select **Modular**.

For a model named `model` and a function named `fn1`, the code generator creates files with modular file packaging.

Modular File Packaging of Files for GRT System Target or ERT System Target

File	Folder	Contents
<code>model.c</code>	<code>model_target_rtw</code>	Calls to the function.
<code>model.h</code>	<code>model_target_rtw</code>	This header file includes declarations and header files for the function, including <code>fn1.h</code> and <code>fn1_private.h</code> .
<code>fn1.c</code>	<code>model_target_rtw</code>	Code for the function.
<code>fn1.h</code>	<code>slprj/target/_sharedutils</code>	This header file contains the <code>fn1</code> function prototype declaration. This header file is included in the code generated for the Function Caller blocks associated with the function. The <code>fn1.h</code> file placement is not affected by the Code Generation > Interface > Shared code placement parameter value.
<code>fn1_private.h</code>	<code>model_target_rtw</code>	This header file includes declarations and header files for the function, including <code>fn1.h</code> .

If you have an Embedded Coder license, set the system target file to `ert.tlc`, and set the **File packaging format** parameter to **Compact** or **Compact (with separate data file)**, the code generator creates files with compact file packaging.

Compact File Packaging of Files for ERT System Target

File	Folder	Contents
<code>model.c</code>	<code>model_ert_rtw</code>	Calls to the function and code for the function.

File	Folder	Contents
model.h	model_ert_rtw	This header file includes declarations and header files for the function, including fn1.h.
fn1.h	slprj\ert _sharedutils	This header file contains the fn1 function prototype declaration. This header file is included in the code generated for the Function Caller blocks associated with the function. The fn1.h file placement is not affected by the Code Generation > Interface > Shared code placement parameter value.

For more information, see “Generate Code for Functions and Callers” on page 7-6.

More About

- “Design Models for Generated Embedded Code Deployment”
- “Generate Code for Functions and Callers” on page 7-6
- “Entry-Point Functions and Scheduling”
- “Configure AUTOSAR Client-Server Communication”
- “Simulink Functions in Models”
- “Simulink Functions in Referenced Models”
- “Simulink Functions”

Generate Code for Functions and Callers

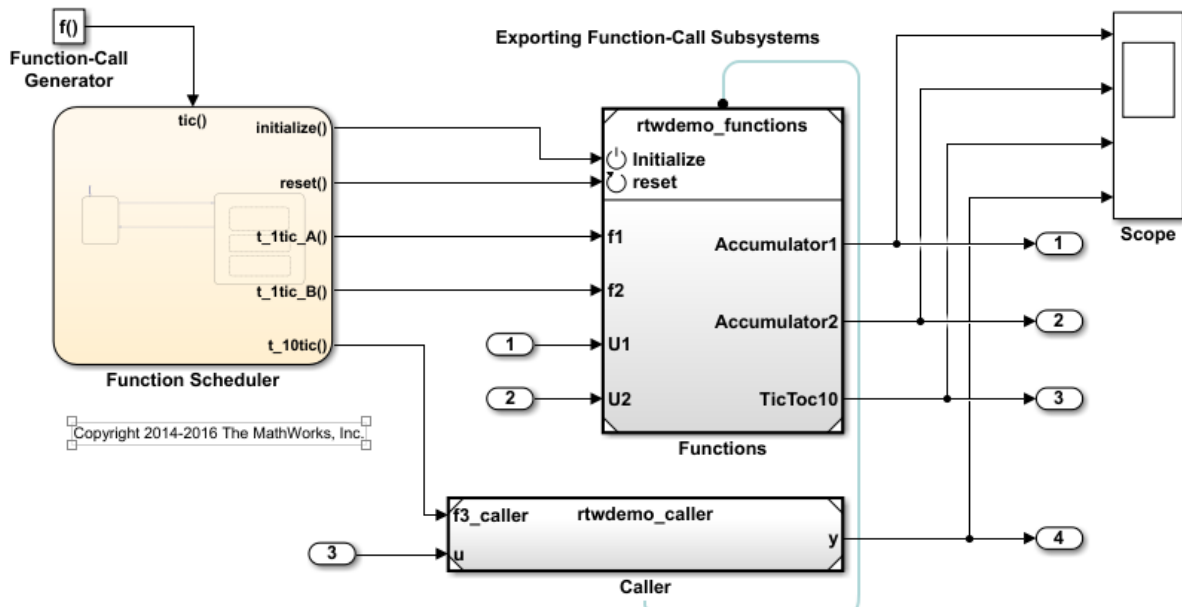
In this section...

“Generate Code for the Function Definition” on page 7-6

“Generate Code for the Caller Definition” on page 7-8

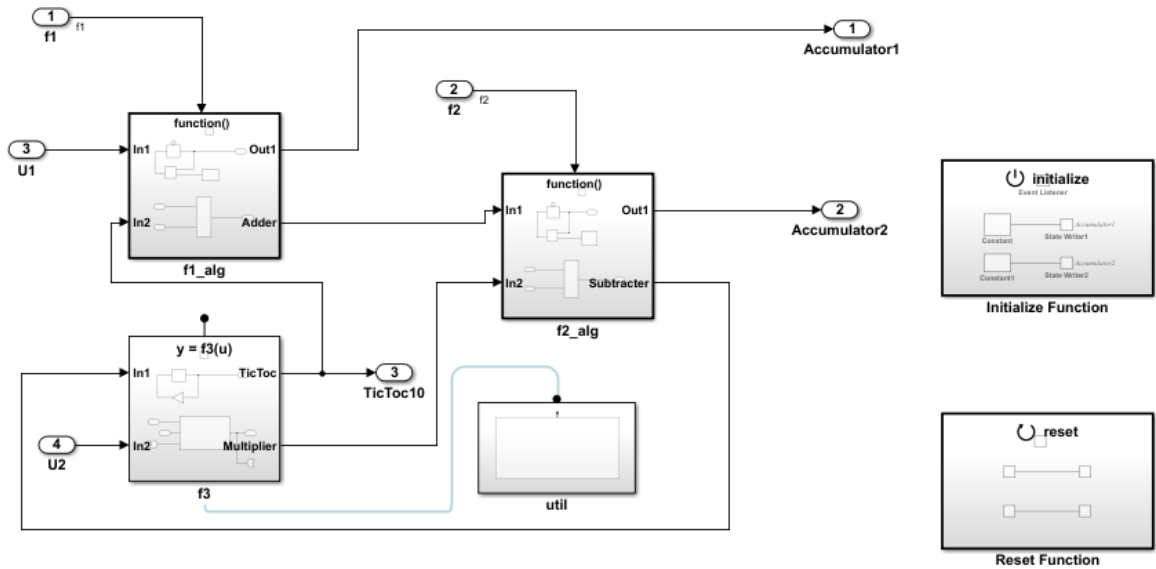
This example shows how the code generator translates Simulink Function blocks and Function Caller blocks into C code.

At the command prompt, type `rtwdemo_export_functions`. This model uses Stateflow software, but this example reviews only the code generated from the referenced models.



Generate Code for the Function Definition

- 1 Double click `rtwdemo_functions`. The Simulink Function block is the `f3` subsystem defined as $y = f3(u)$.



Copyright 2014-2016 The MathWorks, Inc.

2 In the model window, press **Ctrl+B**.

The code generator creates `rtwdemo_functions.c`. This file contains the function definition and initialization.

- The initialization function is:

```
void f3_Init(void)
{
    /* InitializeConditions for UnitDelay: '<S5>/Delay' */
    rtDWork.Delay_DSTATE = 1;
}
```

- The primary function is:

```
/* Output and update for Simulink Function: '<Root>/f3' */
void f3(real_T rtu_u, real_T *rty_y)
{
    /* Outport: '<Root>/TicToc10' incorporates:
    * UnitDelay: '<S5>/Delay' */
```

```

    */
    rtY.TicToc10 = rtDWork.Delay_DSTATE;

    /* Gain: '<S5>/Gain' */
    rtDWork.Delay_DSTATE = (int8_T)(int32_T)-(int32_T)rtY.TicToc10;

    /* FunctionCaller: '<S5>/Function Caller' incorporates:
    *   Inport: '<Root>/U2'
    *   SignalConversion: '<S5>/TmpLatchAtIn1Outputport1'
    *   SignalConversion: '<S5>/TmpSignal ConversionAtuOutputport1'
    */
    adder_h(rtB.Subtract, rtU.U2, rtu_u, rtB.FunctionCaller);

    /* SignalConversion: '<S5>/TmpSignal ConversionAtyInport1' */
    *rty_y = rtB.FunctionCaller;
}
/* Output and update for Simulink Function: '<S6>/adder' */
void adder_h(real_T rtu_u1, real_T rtu_u2, real_T rtu_u3, real_T *rty_y)
{
    /* SignalConversion: '<S7>/TmpSignal ConversionAtyInport1' incorporates:
    *   SignalConversion: '<S7>/TmpSignal ConversionAtu1Outputport1'
    *   SignalConversion: '<S7>/TmpSignal ConversionAtu2Outputport1'
    *   SignalConversion: '<S7>/TmpSignal ConversionAtu3Outputport1'
    *   Sum: '<S7>/Sum'
    */
    *rty_y = (rtu_u1 + rtu_u2) + rtu_u3;
}

```

- The shared header file, `f3.h`, contains the primary function prototype declaration.

```

/* Shared type includes */
#include "rtwtypes.h"

extern void f3(real_T rtu_u, real_T *rty_y);

```

Generate Code for the Caller Definition

- 1 On the `rtwdemo_export_functions` model, click `rtwdemo_caller`.
- 2 Press **Ctrl+B**.

The code generator creates the files `rtwdemo_caller.h` and `rtwdemo_caller.c` in the folder `rtwdemo_caller_ert_rtw`.

rtwdemo_caller.h includes the shared header file, f3.h, which contains the function prototype declaration.

rtwdemo_caller.c calls the function f3.

```
/* Output function for RootInportFunctionCallGenerator: '  
  <Root>/RootFcnCall_InsertedFor_t_10tic_at_outport_1' */  
void rtwdemo_caller_t_10tic(const real_T *rtu_u, real_T *rty_y)  
{  
  /* RootInportFunctionCallGenerator: '  
    <Root>/RootFcnCall_InsertedFor_t_10tic_at_outport_1' incorporates:  
    * SubSystem: '<Root>/Subsystem'  
    */  
  /* FunctionCaller: '<S1>/Function Caller' */  
  f3(*rtu_u, rty_y);  
}
```

More About

- “Design Models for Generated Embedded Code Deployment”
- “Modeling Functions and Callers for Code Generation” on page 7-2
- “Entry-Point Functions and Scheduling”
- “Configure AUTOSAR Client-Server Communication”

Referenced Models in Simulink Coder

- “Code Generation of Referenced Models” on page 8-2
- “Generate Code for Referenced Models” on page 8-4
- “Code Generation Folder Structure for Model Reference Targets” on page 8-15
- “Configure Referenced Models” on page 8-16
- “Build Model Reference Targets” on page 8-17
- “Simulink Coder Model Referencing Requirements” on page 8-18
- “Storage Classes for Signals Used with Model Blocks” on page 8-22
- “Inherited Sample Time for Referenced Models” on page 8-25
- “Customize Library File Suffix and File Type” on page 8-27
- “Reusable Code and Referenced Models” on page 8-28
- “Simulink Coder Model Referencing Limitations” on page 8-32

Code Generation of Referenced Models

This section describes model referencing considerations that apply specifically to code generation by the Simulink Coder. This section assumes that you understand referenced models and related terminology and requirements, as described in “Overview of Model Referencing” and associated topics.

When generating code for a referenced model hierarchy, the code generator produces a stand-alone executable for the top model, and a library module called a *model reference target* for each referenced model. When the code executes, the top executable invokes the model reference targets to compute the referenced model outputs. Model reference targets are sometimes called *Simulink Coder targets*.

Be careful not to confuse a model reference target (Simulink Coder target) with other types of targets:

- Target hardware — A platform for which the Simulink Coder software generates code
- System target — A file that tells the Simulink Coder software how to generate code for particular purpose
- Rapid Simulation target (RSim) — A system target file supplied with the Simulink Coder product
- Simulation target — A MEX-file that implements a referenced model that executes with Simulink Accelerator™ software

The code generator places the code for the top model of a hierarchy in the code generation folder and places the code for referenced models in an `slprj` folder in the code generation folder. Subfolders in `slprj` provide separate places for different types of files. See “Code Generation Folder Structure for Model Reference Targets” on page 8-15 for details.

By default, the product uses *incremental code generation*. When generating code, it compares structural checksums of referenced model files with the generated code files to determine whether to regenerate model reference targets. To control when rebuilds occur, use the configuration parameter **Model Referencing > Rebuild**. For details, see “Rebuild”.

In addition to incremental code generation, the Simulink Coder software uses *incremental loading*. The code for a referenced model is not loaded into memory until the code for its parent model executes and needs the outputs of the referenced model. The

product then loads the referenced model target and executes. Once loaded, the target remains in memory until it is no longer used.

Most code generation considerations are the same whether or not a model includes referenced models: the code generator handles the details automatically insofar as possible. This chapter describes topics that you may need to consider when generating code for a model reference hierarchy.

If you have a Embedded Coder license, custom targets must declare themselves to be model reference compliant if they need to support Model blocks. For more information, see “Support Model Referencing” on page 37-83.

Generate Code for Referenced Models

In this section...

“About Generating Code for Referenced Models” on page 8-4

“Create and Configure the Subsystem” on page 8-4

“Convert Model to Use Model Referencing” on page 8-7

“Generate Model Reference Code for a GRT Target” on page 8-11

“Work with Code Generation Folders” on page 8-13

About Generating Code for Referenced Models

To generate code for referenced models, you

- 1 Create a subsystem in an existing model.
- 2 Convert the subsystem to a referenced model (Model block).
- 3 Call the referenced model from the top model.
- 4 Generate code for the top model and referenced model.
- 5 Explore the generated code and the code generation folder.

You can accomplish some of these tasks automatically with a function called `Simulink.Subsystem.convertToModelReference`.

Create and Configure the Subsystem

In the first part of this example, you define a subsystem for the `vdp` example model, set configuration parameters for the model, and use the `Simulink.Subsystem.convertToModelReference` function to convert it into two new models — the top model (`vdptop`) and a referenced model `vdpmultRM` containing a subsystem you created (`vdpmult`).

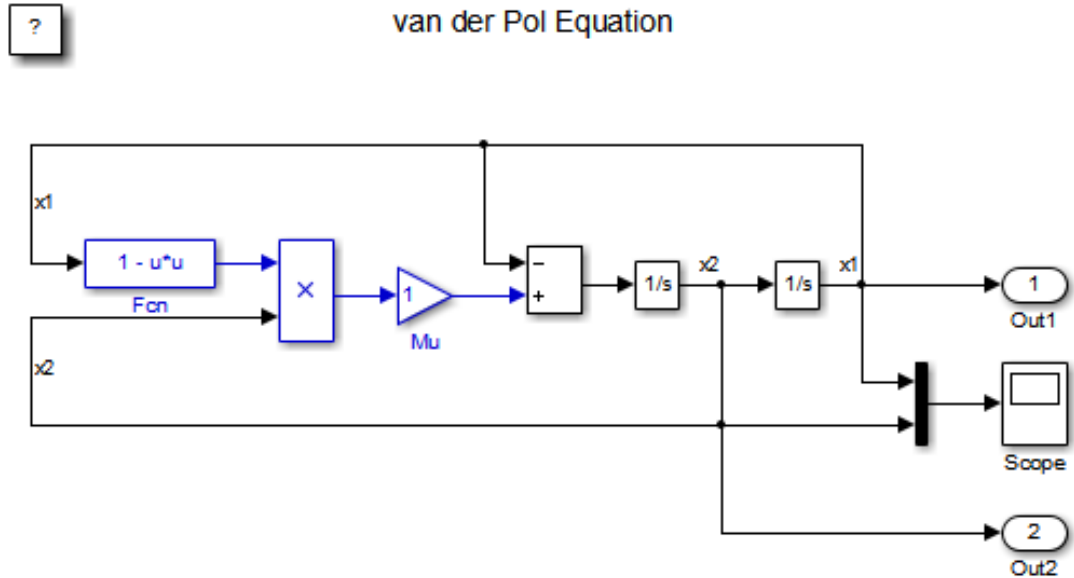
- 1 In the MATLAB Command Window, create a new working folder wherever you want to work and `cd` into it:

```
mkdir mrexample
cd mrexample
```

- 2 Open the `vdp` example model by typing:

vdp

- 3 Drag a box around the three blocks outlined in blue below:



- 4 Choose **Create Subsystem from Selection** from the **Diagram > Subsystem & Model Reference** menu.

A subsystem block replaces the selected blocks.

- 5 If the new subsystem block is not where you want it, move it to a preferred location.
 6 Rename the block vdpmult.
 7 Right-click the vdpmult block and select **Block Parameters (Subsystem)**.

The **Function Block Parameters** dialog box appears.

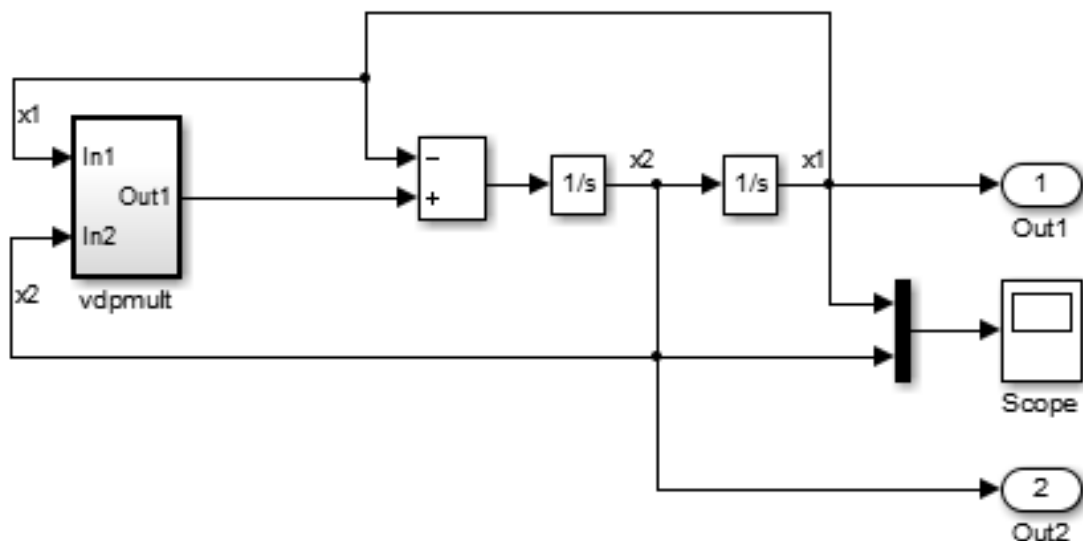
- 8 In the **Function Block Parameters** dialog box, select **Treat as atomic unit**, then click **OK**.

The border of the vdpmult subsystem thickens to indicate that it is now atomic. An atomic subsystem executes as a unit relative to the parent model: subsystem block execution does not interleave with parent block execution. This property makes it

possible to extract subsystems for use as stand-alone models and as functions in generated code.

The block diagram should now appear as follows:

van der Pol Equation



You must set several properties before you can extract a subsystem for use as a referenced model. To set the properties,

- 1 Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
- 2 In the **Model Hierarchy** pane, click the symbol preceding the model name to reveal its components.
- 3 Click **Configuration (Active)** in the left pane.
- 4 In the center pane, select **Solver**.
- 5 In the right pane, under **Solver Options** change the **Type** to **Fixed-step**, then click **Apply**. You must use fixed-step solvers when generating code, although referenced models can use different solvers than top models.

- 6 In the center pane, select **Diagnostics**. In the right pane:
 - a Select the **Data Validity** tab. In the **Signals** area, set **Signal resolution** to **Explicit only**.

Alternatively, if you do not want to use `Simulink.Signal` objects, set **Signal resolution** to **None**.

- b Select the **Connectivity** tab. In the **Buses** area, set **Mux blocks used to create bus signals** to **error**.
- 7 Click **Apply**.

The model now has the properties that model referencing requires.

- 8 In the center pane, click **Model Referencing**. In the right pane, set **Rebuild** to **If any changes in known dependencies detected**. Click **Apply**. This setting prevents code regeneration when it is not required.
- 9 In the `vdpmult` model window, choose **File > Save as**. Save the model as `vdptop` in your working folder. Leave the model open.

Convert Model to Use Model Referencing

In this portion of the example, you use the conversion function `Simulink.SubSystem.convertToModelReference` to extract the subsystem `vdpmult` from `vdptop` and convert `vdpmult` into a referenced model named `vdpmultRM`. To see the complete syntax of the conversion function, type at the MATLAB prompt:

```
help Simulink.SubSystem.convertToModelReference
```

For additional information, type:

```
doc Simulink.SubSystem.convertToModelReference
```

If you want to see an example of `Simulink.SubSystem.convertToModelReference` before using it yourself, type:

```
sldemo_md1ref_conversion
```

Simulink also provides a menu command, **Subsystem & Model Reference > Convert Subsystem to > Referenced Model**, that you can use to convert a subsystem to a referenced model. The command calls

`Simulink.SubSystem.convertToModelReference` with default arguments. For more information, see “Convert a Subsystem to a Referenced Model”.

Extract the Subsystem to a Referenced Model

To use `Simulink.SubSystem.convertToModelReference` to extract `vdpmult` and convert it to a referenced model, type:

```
Simulink.SubSystem.convertToModelReference...  
( 'vdptop/vdpmult', 'vdpmultRM', ...  
'ReplaceSubsystem', true, 'BuildTarget', 'Sim' )
```

This command:

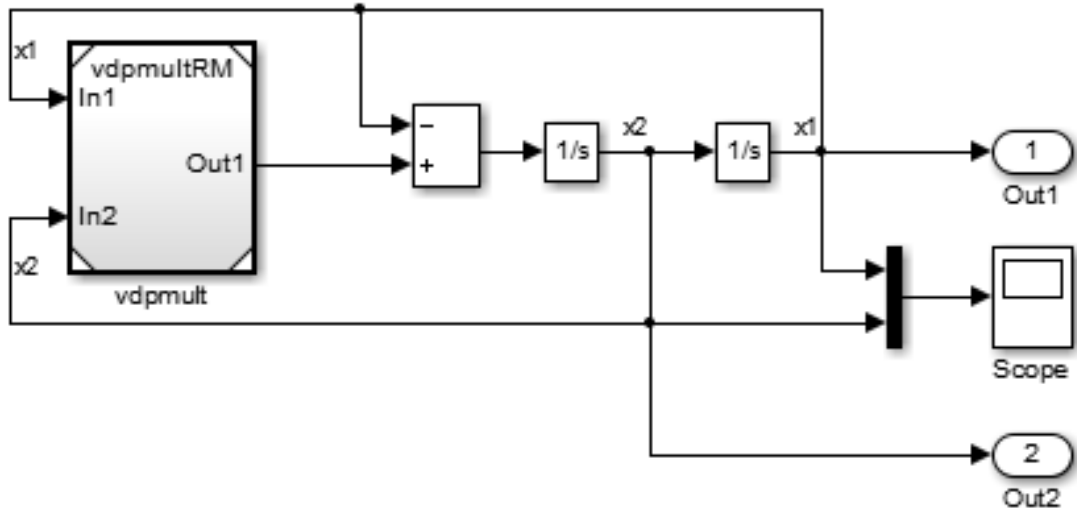
- 1** Extracts the subsystem `vdpmult` from `vdptop`.
- 2** Converts the extracted subsystem to a separate model named `vdpmultRM` and saves the model to the working folder.
- 3** In `vdptop`, replaces the extracted subsystem with a Model block that references `vdpmultRM`.
- 4** Creates a simulation target for `vdptop` and `vdpmultRM`.

The converter prints progress messages and terminates with

```
ans =  
    1
```

The parent model `vdptop` now looks like this:

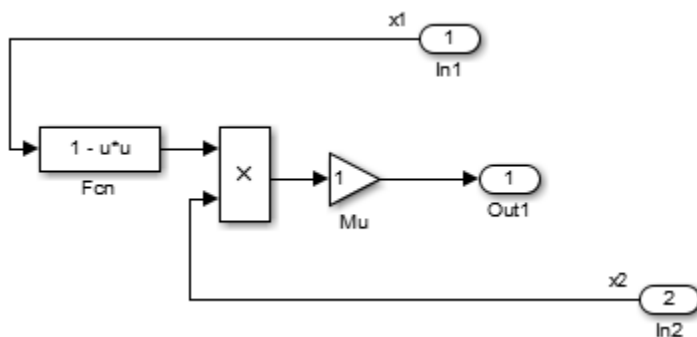
van der Pol Equation



Note the changes in the appearance of the block `vdpmult`. These changes indicate that it is now a Model block rather than a subsystem. As a Model block, it does not have contents of its own: the previous contents now exist in the referenced model `vdpmultRM`, whose name appears at the top of the Model block. Widen the Model block to expose the complete name of the referenced model.

If the parent model `vdptop` had been closed at the time of conversion, the converter would have opened it. Extracting a subsystem to a referenced model does *not* automatically create or change a saved copy of the parent model. To preserve the changes to the parent model, save `vdptop`.

Right-click the Model block `vdpmultRM` and choose **Open** to open the referenced model. The model looks like this:



Files Created and Changed by the Converter

The files in your working folder now consist of the following (not in this order).

File	Description
vdptop model file	Top model that contains a Model block where the <code>vdpmult</code> subsystem was
vdpmultRM model file	Referenced model created for the <code>vdpmult</code> subsystem
vdpmultRM_msf.mexw64	Static library file (Microsoft Windows platforms only). The file extension is system-dependent and may differ. This file executes when the <code>vdptop</code> model calls the Model block <code>vdpmult</code> . When called, <code>vdpmult</code> in turn calls the referenced model <code>vdpmultRM</code> .
/slprj	Folder for generated model reference code

Code for model reference simulation targets is placed in the `slprj/sim` subfolder. Generated code for GRT, ERT, and other Simulink Coder targets is placed in `slprj` subfolders named for those targets. You will inspect some model reference code later in this example. For more information on code generation folders, see “Work with Code Generation Folders” on page 8-13.

Run the Converted Model

Open the Scope block in `vdptop` if it is not visible. In the `vdptop` window, click the **Run** tool or choose **Run** from the **Simulation** menu. The model calls the `vdpmultRM_msf` simulation target to simulate. The output looks like this:



Generate Model Reference Code for a GRT Target

The function `Simulink.SubSystem.convertToModelReference` created the model and the simulation target files for the referenced model `vdpmultRM`. In this part of the example, you generate code for that model and the `vdptop` model, and run the executable you create:

- 1 Verify that you are still working in the `mrexample` folder.
- 2 If the model `vdptop` is not open, open it. Make sure it is the active window.
- 3 Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
- 4 In the **Model Hierarchy** pane, click the symbol preceding the `vdptop` model to reveal its components.
- 5 Click **Configuration (Active)** in the left pane.
- 6 In the center pane, select **Data Import/Export**.
- 7 In the pane, select **Time** and **Output** and *clear* **Data stores**. Click **Apply**.

These settings instruct the model `vdptop` (and later its executable) to log time and output data to MAT-files for each time step.

- 8 Generate GRT code (the default) and an executable for the top model and the referenced model. For example, in the model, press **Ctrl+B**.

The Simulink Coder build process generates and compiles code. The current folder now contains a new file and a new folder:

File	Description
vdptop.exe	The executable created by the build process
vdptop_grt_rtw/	The build folder, containing generated code for the top model

The build process also generated GRT code for the referenced model and placed it in the slprj folder.

To view a model's generated code in **Model Explorer**, the model must be open. To use the **Model Explorer** to inspect the newly created build folder, `vdptop_grt_rtw`:

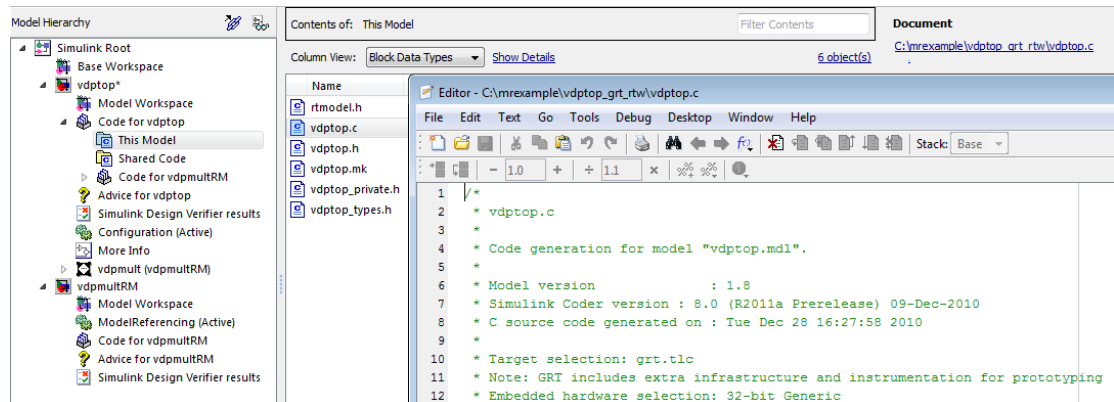
- 1 Open Model Explorer by selecting **Model Explorer** from the model's **View** menu.
- 2 In the **Model Hierarchy** pane, click the symbol preceding the model name to reveal its components.
- 3 Click the symbol preceding **Code** for `vdptop` to reveal its components.
- 4 Directly under **Code** for `vdptop`, click **This Model**.

A list of generated code files for `vdptop` appears in the **Contents** pane:

```
rtmodel.h  
vdptop.c  
vdptop.h  
vdptop.mk  
vdptop_private.h  
vdptop_types.h
```

You can browse code by selecting a file of interest in the **Contents** pane.

To open a file in a text editor, click a filename, and then click the hyperlink that appears in the gray area at the top of the **Document** pane. The figure below illustrates viewing code for `vdptop.c`, in a text editor. Your code may differ.



To view the generated code in the HTML code generation report, see “Generate a Code Generation Report” on page 20-5.

Work with Code Generation Folders

When you view generated code in **Model Explorer**, the files listed in the **Contents** pane can exist either in a build folder or a code generation folder. Model reference code generation folders (located under the `slprj` folder), like build folders, are created in your code generation folder. This process implies certain constraints on when and where model reference targets are built and on how they are accessed.

The models referenced by Model blocks can be stored anywhere. A given top model can include models stored on different file systems or in different folders. The same is not true for the simulation targets derived from these models; under most circumstances, models referenced by a given top model must be set up to simulate and generate model reference target code in a single code generation folder. The top and referenced models can exist anywhere on your path, but the code generation folder is assumed to exist in your current folder.

This means that, if you reference the same model from several top models, each stored in a different folder, you must either

- Always work in the same folder and be sure that the models are on your path
- Allow separate code generation folders, simulation targets, and Simulink Coder targets to be generated in each folder in which you work

The files in such multiple code generation folders are generally quite redundant. Therefore, to minimize code regeneration of referenced models, choose a specific working folder and remain in it for all sessions.

As model reference code generated for Simulink Coder targets as well as for simulation targets is placed in code generation folders, the same considerations as above apply even if you are generating target applications only. That is, code for all models referenced from a given model ends up being generated in the same code generation folder, even if it is generated for different targets and at different times.

Related Examples

- “Specify Instance-Specific Parameter Values for Reusable Referenced Model” on page 14-61

Code Generation Folder Structure for Model Reference Targets

Code for models referenced by using Model blocks is generated in folders under the `slprj` folder in the code generation folder. The next level within `slprj` contains parallel build area subfolders.

The following table lists principal code generation folders and files. In the paths listed, *model* is the name of the model being used as a referenced model, and *target* is the system target file acronym (for example, `grt`, `ert`, `rsim`, and so on).

Folders and Files	Description
<code>slprj/sim/model/</code>	Simulation target files for referenced models
<code>slprj/sim/model/tmwinternal</code>	MAT-files used during code generation
<code>slprj/target/model/ referenced_model_includes</code>	Header files from models referenced by this model
<code>slprj/target/model</code>	Model reference target files
<code>slprj/target/model/tmwinternal</code>	MAT-files used during code generation
<code>slprj/sl_proj.tmw</code>	Marker file
<code>slprj/target/_sharedutils</code>	Utility functions for model reference targets, shared across models
<code>slprj/sim/_sharedutils</code>	Utility functions for simulation targets, shared across models

If you are building code for more than one referenced model within the same code generation folder, the model reference files are added to the existing `slprj` folder.

Configure Referenced Models

Minimize occurrences of algebraic loops by selecting the **Minimize algebraic loop occurrences** parameter on the **Model Reference** pane. The setting of this option affects only generation of code from the model. See “Platform Options for Development and Deployment” on page 10-2 in the Simulink Coder documentation for information on how this option affects code generation. For more information, see “Model Blocks and Direct Feed through”.

Use the **Integer rounding mode** parameter on your model's blocks to simulate the rounding behavior of the C compiler that you intend to use to compile code generated from the model. This setting appears on the **Signal Attributes** pane of the parameter dialog boxes of blocks that can perform signed integer arithmetic, such as the **Product** and **n-D Lookup Table** blocks.

For most blocks, the value of **Integer rounding mode** completely defines rounding behavior. For blocks that support fixed-point data and the Simplest rounding mode, the value of **Signed integer division rounds to** also affects rounding. For details, see “Precision”.

When models contain Model blocks, all models that they reference must be configured to use identical hardware settings. For information on the **Model Referencing** pane options, see “Model Configuration Parameters: Model Referencing” and “Configuration Parameter Requirements”.

Build Model Reference Targets

By default, the Simulink engine rebuilds simulation targets before the Simulink Coder software generates model reference targets. You can change the rebuild criteria or specify when the engine rebuilds targets. For more information, see “Rebuild”.

The Simulink Coder software generates a model reference target directly from the Simulink model. The product automatically generates or regenerates model reference targets, for example, when they require an update.

You can command the Simulink and Simulink Coder products to generate a simulation target for an Accelerator mode referenced model, and a model reference target for a referenced model, by executing the `slbuild` command with arguments in the MATLAB Command Window.

The Simulink Coder software generates only one model reference target for all instances of a referenced model. See “Reusable Code and Referenced Models” on page 8-28 for details.

Reduce Change Checking Time

You can reduce the time that the Simulink and Simulink Coder products spend checking whether simulation targets and model reference targets need to be rebuilt by setting configuration parameter values as follows:

- In the top model, consider setting the model configuration parameter **Model Referencing > Rebuild** to **If any changes in known dependencies detected**. (See “Rebuild”.)
- In all referenced models throughout the hierarchy, set the configuration parameter **Diagnostics > Data Validity > Signal resolution** to **Explicit only** or **None**. (See “Signal resolution”.)

These parameter values exist in a referenced model's configuration set, not in the individual Model block. Setting either value for an instance of a referenced model, sets it for all instances of that model.

Simulink Coder Model Referencing Requirements

A model reference hierarchy must satisfy various Simulink Coder requirements, as described in this section. In addition to these requirements, a model referencing hierarchy to be processed by the Simulink Coder software must satisfy:

- The Simulink requirements listed in:
 - “Configuration Requirements for All Referenced Model Simulation”
 - “Model Structure Requirements”
- The Simulink limitations listed in “Limitations on All Model Referencing”
- The Simulink Coder limitations listed in “Simulink Coder Model Referencing Limitations” on page 8-32

Configuration Parameter Requirements

A referenced model uses a configuration set in the same way a top model does, as described in “Manage a Configuration Set”. By default, every model in a hierarchy has its own configuration set, which it uses in the same way that it would if the model executed independently.

Because each model can have its own configuration set, configuration parameter values can be different in different models. Furthermore, some parameter values are intrinsically incompatible with model referencing. The response of the Simulink Coder software to an inconsistent or unusable configuration parameter depends on the parameter:

- Where an inconsistency has no significance, the product ignores or resolves the inconsistency without posting a warning.
- Where a nontrivial and possibly acceptable solution exists, the product resolves the conflict silently; resolves it with a warning; or generates an error.
- If an acceptable resolution is not possible, the product generates an error. You must then change parameter values to eliminate the problem.

When a model reference hierarchy contains many referenced models that have incompatible parameter values, or a changed parameter value must propagate to many referenced models, manually eliminating all configuration parameter incompatibilities can be tedious. You can control or eliminate such overhead by using configuration

references to assign an externally-stored configuration set to multiple models. See “Manage a Configuration Reference” for details.

The following tables list configuration parameters that can cause problems if set in certain ways, or if set differently in a referenced model than in a parent model. Where possible, the Simulink Coder software resolves violations of these requirements automatically, but most cases require changes to the parameters in some or all models.

Configuration Requirements for Model Referencing with All System Targets

Dialog Box Pane	Option	Requirement
Solver	Start time	Some system targets require the start time of all models to be zero.
Hardware Implementation	All options	Values must be the same for top and referenced models.
Code Generation	System target file	Must be the same for top and referenced models.
	Language	Must be the same for top and referenced models.
	Generate code only	Must be the same for top and referenced models.
Symbols	Maximum identifier length	Cannot be longer for a referenced model than for its parent model.
Interface	Code replacement library	Must be the same for top and referenced models.
	C API options	The C API check boxes must be the same for top and referenced models.
	ASAP2 interface	Can be on or off in a top model, but must be off in a referenced model. If it is not, the Simulink Coder software temporarily sets it to off during code generation.

Configuration Requirements for Model Referencing with ERT System Targets (Requires Embedded Coder License)

Dialog Box Pane	Option	Requirement
All Parameters tab	Ignore custom storage classes	Must be the same for top and referenced models.
Symbols	Global variables Global types Subsystem methods Local temporary variables Constant macros	\$R token must appear.
	Signal naming	Must be the same for top and referenced models.
	M-function	If specified, must be the same for top and referenced models.
	Parameter naming	Must be the same for top and referenced models.
	#define naming	Must be the same for top and referenced models.
Interface	Support floating-point numbers	Must be the same for both top and referenced models
	Support non-finite numbers	If <code>Off</code> for top model, must be <code>Off</code> for referenced models.
	Support complex numbers	If <code>Off</code> for top model, must be <code>Off</code> for referenced models.
	Suppress error status in real-time model	If <code>On</code> for top model, must be <code>On</code> for referenced models.
Code Placement	Use owner from data object for data definition placement	Must be the same for top and referenced models.
	Signal display level	Must be the same for top and referenced models.
	Parameter tune level	Must be the same for top and referenced models.

Naming Requirements

Within a model that uses model referencing, names of the constituent models can not collide. When you generate code from a model that uses model referencing, the **Maximum identifier length** parameter must be large enough to accommodate the root model name and the name-mangling text. A code generation error occurs if **Maximum identifier length** is not large enough.

When a name conflict occurs between a symbol within the scope of a higher-level model and a symbol within the scope of a referenced model, the symbol from the referenced model is preserved. Name mangling is performed on the symbol from the higher-level model.

Embedded Coder Naming Requirements

The Embedded Coder product lets you control the formatting of generated symbols in much greater detail. When generating code with an ERT target from a model that uses model referencing:

- The **\$R** token must be included in the **Identifier format control** parameter specifications (in addition to the **\$M** token) except for **Shared utilities**.
- The **Maximum identifier length** must be large enough to accommodate full expansions of the **\$R** and **\$M** tokens.

See “Model Configuration Parameters: Code Generation Symbols” for more information.

Custom Target Requirements

If you have an Embedded Coder license, a custom target must meet various requirements to support model referencing. For details, see “Support Model Referencing” on page 37-83.

Storage Classes for Signals Used with Model Blocks

Models containing Model blocks can use signals of storage class `Auto` without restriction. However, when you declare signals to be global, you must be aware of how the signal data will be handled.

A global signal is a signal with a storage class other than `Auto`:

- `ExportedGlobal`
- `ImportedExtern`
- `ImportedExternPointer`
- `Custom`

The above are distinct from `SimulinkGlobal` signals, which are treated as test points with `Auto` storage class.

Global signals are declared, defined, and used as follows:

- An `extern` declaration is generated by all models that use a given global signal.

As a result, if a signal crosses a Model block boundary, the top model and the referenced model both generate `extern` declarations for the signal.
- For an exported signal, the top model is responsible for defining (allocating memory for) the signal, whether or not the top model itself uses the signal.
- Global signals used by a referenced model are accessed directly (as global memory). They are not passed as arguments to the functions that are generated for the referenced models.

Custom storage classes also follow the above rules. However, certain custom storage classes are not currently supported for use with model reference. See “Custom Storage Class Limitations” for details.

Storage Classes for Parameters Used with Model Blocks

Storage classes are supported for both simulation and code generation, and all except `Auto` are tunable. The supported storage classes thus include

- `SimulinkGlobal`
- `ExportedGlobal`

- `ImportedExtern`
- `ImportedExternPointer`
- `Custom`

Note the following restrictions on parameters in referenced models:

- Tunable parameters are not supported for noninlined S-functions.
- Tunable parameters set using the Model Parameter Configuration dialog box are ignored.

Note the following considerations concerning how global tunable parameters are declared, defined, and used in code generated for targets:

- A global tunable parameter is a parameter in the base workspace with a storage class other than `Auto`.
- An `extern` declaration is generated by all models that use a given parameter.
- If a parameter is exported, the top model is responsible for defining (allocating memory for) the parameter (whether it uses the parameter or not).
- Global parameters are accessed directly (as global memory). They are not passed as arguments to the functions that are generated for the referenced models.
- Symbols for `SimulinkGlobal` parameters in referenced models are generated using unstructured variables (`rtP_xxx`) instead of being written into the `model_P` structure. This is so that each referenced model can be compiled independently.

Certain custom storage classes for parameters are not currently supported for model reference. See “Custom Storage Class Limitations” for details.

Parameters used as Model block arguments must be defined in the referenced model's workspace. See “Parameterize Instances of a Reusable Referenced Model” in the Simulink documentation for specific details.

Signal Name Mismatches Across Model Reference Boundary

Within a parent model, the name and storage class for a signal entering or leaving a Model block might not match those of the signal attached to the root inport or outport within that referenced model. Because referenced models are compiled independently without regard to a parent model, they cannot adapt to the possible variations in how parent models label and store signals.

The Simulink Coder software accepts all cases where input and output signals in a referenced model have `Auto` storage class. When such signals are test pointed or are global, as described above, certain restrictions apply. The following table describes how mismatches in signal labels and storage classes between parent and referenced models are handled:

Relationships of Signals and Storage Classes Across Model Reference Boundary

Referenced Model	Parent Model	Signal Passing Method	Signal Mismatch Checking
Auto	Any storage class	Function argument	None
SimulinkGlobal or resolved to Signal Object	Any storage class	Function argument	Signal label mismatch
Global	Auto or SimulinkGlobal	Global variable	Signal label mismatch
Global	Global	Global variable	Labels and storage classes must be identical (else error)

To summarize, the following signal resolution rules apply to code generation:

- If the storage class of a root input or output signal in a referenced model is `Auto` (or is `SimulinkGlobal`), the signal is passed as a function argument.
 - When such a signal is `SimulinkGlobal` or resolves to a `Simulink.Signal` object, the **Signal label mismatch** diagnostic is applied.
- If a root input or output signal in a referenced model is global, it is communicated by using direct memory access (global variable). In addition,
 - If the corresponding signal in the parent model is also global, the names and storage classes must match exactly.
 - If the corresponding signal in the parent model is not global, the **Signal label mismatch** diagnostic is applied.

You can set the **Signal label mismatch** diagnostic to `error`, `warning`, or `none` in the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box.

Inherited Sample Time for Referenced Models

See “Sample Times for Model Referencing” in the Simulink documentation for information about Model block sample time inheritance. In generated code, you can control inheriting sample time by using `ssSetModelReferenceSampleTimeInheritanceRule` in different ways:

- An S-function that precludes inheritance: If the sample time is used in the S-function's run-time algorithm, then the S-function precludes a model from inheriting a sample time. For example, consider the following `mdlOutputs` code:

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    const real_T *u = (const real_T*)
        ssGetInputPortSignal(S,0);
    real_T *y = ssGetOutputPortSignal(S,0);
    y[0] = ssGetSampleTime(S,tid) * u[0];
}
```

This `mdlOutputs` code uses the sample time in its algorithm, and the S-function therefore should specify

```
ssSetModelReferenceSampleTimeInheritanceRule
(S, DISALLOW_SAMPLE_TIME_INHERITANCE);
```

- An S-function that does not preclude Inheritance: If the sample time is only used for determining whether the S-function has a sample hit, then it does not preclude the model from inheriting a sample time. For example, consider the `mdlOutputs` code from the S-function example `sfun_multirate.c`:

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    InputRealPtrsType enablePtrs;
    int *enabled = ssGetIWork(S);

    if (ssGetInputPortSampleTime
        (S,ENABLE_IPORT)==CONTINUOUS_SAMPLE_TIME &&
        ssGetInputPortOffsetTime(S,ENABLE_IPORT)==0.0) {
        if (ssIsMajorTimeStep(S) &&
            ssIsContinuousTask(S,tid)) {
            enablePtrs =
                ssGetInputPortRealSignalPtrs(S,ENABLE_IPORT);
            *enabled = (*enablePtrs[0] > 0.0);
        }
    }
}
```

```
    } else {
        int enableTid =
            ssGetInputPortSampleTimeIndex(S,ENABLE_IPORT);
        if (ssIsSampleHit(S, enableTid, tid)) {
            enablePtrs =
                ssGetInputPortRealSignalPtrs(S,ENABLE_IPORT);
            *enabled = (*enablePtrs[0] > 0.0);
        }
    }

    if (*enabled) {
        InputRealPtrsType uPtrs =
            ssGetInputPortRealSignalPtrs(S,SIGNAL_IPORT);
        real_T signal = *uPtrs[0];
        int i;

        for (i = 0; i < NOUTPUTS; i++) {
            if (ssIsSampleHit(S,
                ssGetOutputPortSampleTimeIndex(S,i), tid)) {
                real_T *y = ssGetOutputPortRealSignal(S,i);
                *y = signal;
            }
        }
    }
} /* end mdlOutputs */
```

The above code uses the sample times of the block, but only for determining whether there is a hit. Therefore, this S-function should set

```
ssSetModelReferenceSampleTimeInheritanceRule
(S, USE_DEFAULT_FOR_DISCRETE_INHERITANCE);
```

Customize Library File Suffix and File Type

You can control the library file suffix and file type extension that the Simulink Coder code generator uses to name generated model reference libraries. Use the model configuration parameter `TargetLibSuffix` to specify the scheme for the suffix and extension. The scheme must include a period (.). If you do not set this parameter, the Simulink Coder software names the libraries as follows:

- On Windows systems, *model_rtwlib.lib*
- On UNIX or Linux[®] systems, *model_rtwlib.a*

Reusable Code and Referenced Models

Models that employ model referencing might require special treatment when generating and using reusable code. The following sections identify general restrictions and discuss how reusable functions with inputs or outputs connected to a referenced model's root Inport or Outport blocks can affect code reuse.

General Considerations

You can generate code for subsystems that contain referenced models using the same procedures and options described in “Code Generation of Subsystems” on page 6-2. However, the following restrictions apply to such builds:

- A top model that uses single-tasking mode and that has a referenced model that uses multi-tasking mode executes for blocks with the different rates that are not connected. However, you get an error if the blocks with different rates are connected by Rate Transition block (inserted either manually or by Simulink).
- ERT S-functions do not support subsystems that contain a continuous sample time.
- The Simulink Coder S-function target is not supported.
- The Tunable parameters table (set by using the Model Parameter Configuration dialog box) is ignored; to make parameters tunable, you must define them as Simulink parameter objects in the base workspace.
- All other parameters are inlined into the generated code and S-function.

Note You can generate subsystem code using any target configuration available in the System Target File Browser. However, if the S-function target is selected, **Build This Subsystem** and **Build Selected Subsystem** behaves identically to **Generate S-Function**. (See “Automate S-Function Generation with S-Function Builder” on page 24-22.)

Code Reuse and Model Blocks with Root Inport or Outport Blocks

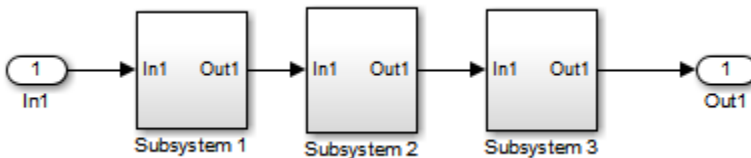
Reusable functions with inputs or outputs connected to a referenced model's root Inport or Outport block can affect code reuse. This means that code for certain atomic subsystems cannot be reused in a model reference context the same way it is reused in a standalone model.

For example, suppose you create the following subsystem and make the following changes to the subsystem's block parameters:

- Select **Treat as an atomic unit**
- Go to the **Code Generation** tab and set **Function packaging** to **Reusable function**



Suppose you then create the following model, which includes three instances of the preceding subsystem.



With the configuration parameter **Default parameter behavior** set to **Inlined** in this stand-alone model, the code generator can optimize the code by generating a single copy of the function for the reused subsystem, as shown below.

```

void reuse_subsys1_Subsystem1(
    real_T rtu_0,
    rtB_reuse_subsys1_Subsystem1 *localB)
{
    /* Gain: '<S1>/Gain' */
    localB->Gain_k = rtu_0 * 3.0;
}
  
```

When generated as code for a Model block (into an `slprj` folder in the code generation folder), the subsystems have three different function signatures:

```

/* Output and update for atomic system: '<Root>/Subsystem1' */
void reuse_subsys1_Subsystem1(const real_T *rtu_0,
  
```

```

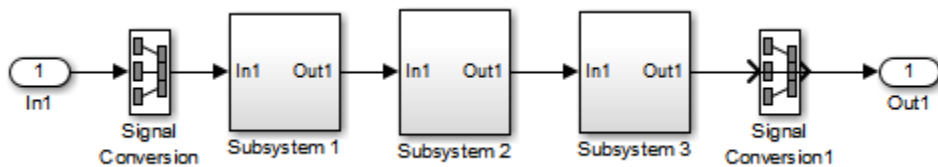
rtB_reuse_subsys1_Subsystem1
 *localB)
{
  /* Gain: '<S1>/Gain' */
  localB->Gain_w = (*rtu_0) * 3.0;
}

/* Output and update for atomic system: '<Root>/Subsystem2' */
void reuse_subsys1_Subsystem2(real_T rtu_In1,
rtB_reuse_subsys1_Subsystem2
 *localB)
{
  /* Gain: '<S2>/Gain' */
  localB->Gain_y = rtu_In1 * 3.0;
}

/* Output and update for atomic system: '<Root>/Subsystem3' */
void reuse_subsys1_Subsystem3(real_T rtu_In1, real_T *rty_0)
{
  /* Gain: '<S3>/Gain' */
  (*rty_0) = rtu_In1 * 3.0;
}

```

One way to make all the function signatures the same for code reuse, is to insert Signal Conversion blocks. Place one between the Inport and Subsystem1 and another between Subsystem3 and the Output of the referenced model.



The result is a single reusable function:

```

void reuse_subsys2_Subsystem1(real_T rtu_In1,
                             rtB_reuse_subsys2_Subsystem1 *localB)
{
  /* Gain: '<S1>/Gain' */
  localB->Gain_g = rtu_In1 * 3.0;
}

```

You can achieve the same result (reusable code) with only one Signal Conversion block. You can omit the Signal Conversion block connected to the Inport block if you select the **Pass fixed-size scalar root inputs by value** check box at the bottom of the **Model Referencing** pane of the Configuration Parameters dialog box. When you do this, you still need to insert a Signal Conversion block before the Outport block.

Simulink Coder Model Referencing Limitations

The following Simulink Coder limitations apply to model referencing. In addition to these limitations, a model reference hierarchy used for code generation must satisfy:

- The Simulink requirements listed in:
 - “Configuration Requirements for All Referenced Model Simulation”
 - “Model Structure Requirements”
- The Simulink limitations listed in “Model Referencing Limitations”.
- The Simulink Coder requirements applicable to the code generation target, as listed in “Configuration Parameter Requirements” on page 8-18.

Customization Limitations

- The code generator ignores custom code settings in the Configuration Parameters dialog box and custom code blocks when generating code for a referenced model.
- Data type replacement is not supported for simulation target code generation of referenced models.
- Simulation targets do not include Stateflow target custom code.
- If you have an Embedded Coder license, some restrictions exist on grouped custom storage classes in referenced models. For details, see “Custom Storage Class Limitations”.

Data Logging Limitations

- To Workspace blocks, Scope blocks, and all types of runtime display, such as the display of port values and signal values, are ignored when the Simulink Coder software generates code for a referenced model. The resulting code is the same as if the constructs did not exist.
- Code generated for referenced models cannot log data to MAT-files. If data logging is enabled for a referenced model, the Simulink Coder software disables the option before code generation and re-enables it afterwards.
- If you log states for a model that contains referenced models, the ordering of the states in the output is determined by block sorted order, and might not match between simulation output and generated code MAT-file logging output.

State Initialization Limitation

When a top model uses the **Data Import/Export > Initial state** parameter in the Configuration Parameters dialog box to specify initial conditions, the Simulink Coder software does not initialize the discrete states of the referenced models during code generation.

Reusability Limitations

If a referenced model used for code generation has any of the following properties, the model must specify the configuration parameter **Model Referencing > Total number of instances allowed per top model** as **One**, and no other instances of the model can exist in the hierarchy. If you do not set the parameter to **One**, or more than one instance of the model exists in the hierarchy, an error occurs. The properties are:

- The model references another model which has been set to single instance
- The model contains a state or signal with non-auto storage class
- The model uses any of the following Stateflow constructs:
 - Machine-parented data
 - Machine-parented events
 - Stateflow graphical functions
- The model contains a subsystem that is marked as function
- The model contains an S-function that is:
 - Inlined but has not set the option `SS_OPTION_WORKS_WITH_CODE_REUSE`
 - Not inlined
- The model contains a function-call subsystem that:
 - Has been forced by the Simulink engine to be a function
 - Is called by a wide signal

For more information about **Total number of instances allowed per top model**, see “Total number of instances allowed per top model”.

S-Function Limitations

- If a referenced model contains an S-function that should be inlined using a Target Language Compiler file, the S-function must use the `ssSetOptions` macro to set the `SS_OPTION_USE_TLC_WITH_ACCELERATOR` option in its `mdlInitializeSizes` method. The simulation target will not inline the S-function unless this flag is set.
- A referenced model cannot use noninlined S-functions generated by the Simulink Coder software.
- The Simulink Coder S-function target does not support model referencing.

For additional information, in the Simulink documentation, see “S-Functions with Referenced Models”.

Simulink Tool Limitations

- Simulink tools that require access to a model's internal data or configuration (including the Model Coverage tool, the Simulink Report Generator product, the Simulink debugger, and the Simulink profiler) have no effect on code generated by the Simulink Coder software for a referenced model, or on the execution of that code. Specifications made and actions taken by such tools are ignored and effectively do not exist.

Subsystem Limitations

- If a subsystem contains Model blocks, you cannot build a subsystem module by right-clicking the subsystem (or by using **Code > C/C++ Code > Build Selected Subsystem**) unless the model is configured to use an ERT target.
- If you generate code for an atomic subsystem as a reusable function, inputs or outputs that connect the subsystem to a referenced model might prevent code reuse, as described in “Reusable Code and Referenced Models” on page 8-28.

Target Limitations

- The Simulink Coder S-function target does not support model referencing.

Other Limitations

- Errors or unexpected behavior can occur if a Model block is part of a cycle, the Model block is a direct feedthrough block, and an algebraic loop results. For details, see “Model Blocks and Direct Feed through”.
- The **External mode** option is not supported. If it is enabled, it is ignored during code generation.
- When a model contains a trigger or enable port, you cannot generate standalone Simulink Coder code or PIL code.

Combined Models in Simulink Coder

- “Combined Models” on page 9-2
- “Use GRT with Reusable Function Packaging to Combine Models” on page 9-4

Combined Models

If you want to combine several models (or several instances of the same model) into a single executable, the code generator offers several options.

The most powerful solution is to use Model blocks. Each instance of a Model block represents another model, called a *referenced model*. For code generation, the referenced model effectively replaces the Model block that references it. For details, see “Overview of Model Referencing” and “Generate Code for Referenced Models” on page 8-4.

When developing embedded systems using the Embedded Coder product, you can interface the code for several models to a common harness program by directly calling the entry points to each model. However, the `ert.tlc` system target file has certain restrictions, relating to embedded processing, that might not be compatible with your application.

The `grt.tlc` system target file with the model parameter **Code interface packaging** set to **Reusable function** is another possible solution. Use it in situations where you want to:

- Deploy more than one instance of a model
- Selectively control calls to more than one instance of a model
- Use dynamic memory allocation
- Include models that employ continuous states
- Log data to multiple files
- Run one of the models in external mode

For more information, see “Use GRT with Reusable Function Packaging to Combine Models” on page 9-4.

To summarize by target, your options are as follows:

System Target File	Support for Combining Multiple Models?
<code>grt.tlc</code> (generic real-time)	Yes (using Model blocks or Code interface packaging set to Reusable function)
<code>ert.tlc</code> (embedded real-time)	Yes
<code>rtwsfcn.tlc</code> (S-function)	No

Control Ownership of Data

If you have an Embedded Coder license, you can specify an *owner* for individual data items such as signals, parameters, and states. The owner of a data item generates the definition (memory allocation and initialization) for the data item. For example, if you apply a custom storage class to a `Simulink.Parameter` object so that it appears as a tunable global variable in the generated code, specify one of the combined models as the owner of the object. The code generated for that model defines the parameter data.

If you use model referencing, you can modularize the generated code and establish clear ownership of data when you work in a team.

If you do not use model referencing, you can prevent generation of duplicate definitions for a data item. For example, suppose you store a `Simulink.Parameter` object in the base workspace and apply the storage class `ExportedGlobal`. If you generate code from two separate models that use the object, each model generates a definition for the corresponding global variable. Instead, you can specify an owner for the object so that only the owner generates a definition.

To specify an owner for a data item:

- 1 Apply a custom storage class to the data item. See “Introduction to Custom Storage Classes”.
- 2 Configure the owner of the data item by specifying the **Owner** custom attribute.
- 3 Select the model configuration parameter **Use owner from data object for data definition placement**.

For more information about controlling ownership and file placement of data definitions and declarations, see “Manage Placement of Data Definitions and Declarations”.

Use GRT with Reusable Function Packaging to Combine Models

Use the GRT system target file with the model configuration parameter **Code interface packaging** set to `Reusable function` to combine models into a single program. (See “Combined Models” on page 9-2 for other ways to combine models into a single program.)

Building a multiple-model executable is fairly straightforward:

- 1 Generate and compile code from each of the models that are to be combined. If you want to include multiple instances of the same model, or if you want model data to be dynamically allocated, set the model parameter **Code interface packaging** to `Reusable function`.
- 2 Combine the makefiles for each of the models into one makefile for creating the final multiple model executable.
- 3 Create a combined simulation engine by modifying a main program, such as `rt_malloc_main.c`, to initialize and call the models.
- 4 Run the combination makefile to link the object files from the models and the main program into an executable.

Share Data Across Models

Use unidirectional signal connections between models. This affects the order in which models are called. For example, if an output signal from `modelA` is used as input to `modelB`, `modelA`'s output computation should be called first.

Timing Issues

You must generate all of the models you are combining with the same solver mode (either all single-tasking or all multitasking.) In addition, if the models employ continuous states, the same solver should be used for all of the models.

Because each model has its own model-specific definition of the `rtModel` data structure, you must use an alternative mechanism to control model execution, as follows:

- The file `rtw/c/src/rtmcmacros.h` provides an `rtModel` API clue that can be used to call the `rt_OneStep` procedure.
- The `rtmcmacros.h` header file defines the `rtModelCommon` data structure, which has the minimum common elements in the `rtModel` structure required to step a model forward one time step.

- The `rtmcsetCommon` macro populates an object of type `rtModelCommon` by copying the respective similar elements in the model's `rtModel` object. Your main routine must create one `rtModelCommon` structure for each model being called by the main routine.
- The main routine will subsequently invoke `rt_OneStep` with a pointer to the `rtModelCommon` structure instead of a pointer to the `rtModel` structure.

If the base rates for the models are not the same, the main program (such as `rt_malloc_main.c`) must set up the timer interrupt to occur at the greatest common divisor rate of the models. The main program calls each model at a time interval.

Data Logging and External Mode Support

A multiple-model program can log data to separate MAT-files for each model.

Only one of the models in a multiple-model program can use external mode.

Configure Model Parameters for Simulink Coder

- “Platform Options for Development and Deployment” on page 10-2
- “Configure Target Hardware” on page 10-3
- “Configure Production Hardware Characteristics” on page 10-13
- “Configure Test Hardware Characteristics” on page 10-14
- “Control the Location for Generated Files” on page 10-15
- “Control Generated Files Location Used for Simulation” on page 10-17
- “Control the Location for Code Generation Files” on page 10-19
- “Override Build Folder Settings for Current Session” on page 10-21

Platform Options for Development and Deployment

When you use Simulink software to create and execute a model, and the code generator to produce C or C++ code, you can consider up to three platforms:

- **MATLAB Host** — The host computer platform that runs MathWorks software during application development.
- **Production Target** — The target hardware platform on which you deploy an application when it is put into production.
- **Test Target** — The platform on which you test an application under development before deployment.

The same platform can serve in two, or possibly three, capacities, but the platforms remain conceptually distinct. Often the MATLAB host and the test target are the same. The production target is usually different from, and less powerful than, the MATLAB host or the test target. Often, the production target can do little more than run a downloaded executable file.

When you use Simulink software to execute a model for which you later generate code, or use the code generator to produce code for deployment on a *production* target, you must provide information about the production hardware board and the compiler that you use with it. The Simulink software uses this information to get bit-true agreement for the results of integer and fixed-point operations performed in simulation and in code generated for the production target. The code generator uses the information to create code that executes with maximum efficiency.

When you generate code for testing on a *test* target, you must provide information about the test hardware board and the compiler that you use with it. The code generator uses this information to create code that provides bit-true agreement for the results of integer and fixed-point operations performed in simulation, in code generated for the production target, and in code generated for the test target. Agreement can result even though the production target and test target can use very different hardware, and the compilers for the two targets can use different defaults, that is, where the C standard does not completely define behavior.

Configure Target Hardware

You can configure model simulation or code generation for a specific hardware board and its device type. For example, you can set the data size, byte ordering, and compiler behavior, such as integer rounding. You can configure two types of target hardware:

- **Production hardware** — Production target hardware and the compiler that you use with it. This information affects simulation and code generation.
- **Test hardware** — Test target hardware and the compiler that you use with it. This information affects only code generation.

Configure production hardware with the Configuration Parameters dialog **Hardware Implementation** pane or from the Configuration Parameters **All Parameters** tab. By default, the Hardware Implementation pane lists **Hardware board**, **Device vendor**, and **Device type** parameter fields only. Unless you have installed hardware support packages, **Hardware board** lists values **None** or **Determine by Code Generation system target file**, and **Get Hardware Support Packages**. After installing a hardware support package, the list also includes the corresponding hardware board name. If you select a hardware board name, parameters for that board appear. To set device details, such as data size and byte ordering, click **Device details**.

Configure test target hardware from the Configuration Parameters **All Parameters** tab. Set **ProdEqTarget** to off, which enables parameters for configuring test target hardware details. Code generated for test target hardware executes in the environment specified by the test target hardware parameters, but behaves as if it were executing in the environment specified for the production target hardware. See “Configure Test Hardware Characteristics” on page 10-14 for details.

If you have used the **Code Generation** pane to specify a **System target file**, and the target file specifies a default microprocessor and its hardware properties, the default and properties appear as initial values in the **Hardware Implementation** pane.

You cannot change parameters that have only one possible value. Parameters that have more than one possible value provide a list of valid values. If you specify hardware properties manually, check carefully that their values are consistent with the system target file. Otherwise, the generated code might fail to compile or execute, or might execute but give incorrect results.

Note: Hardware implementation parameters do not control hardware or compiler behavior. They describe hardware and compiler properties to MATLAB software. The

code generator uses the information to produce code for the platform that runs as efficiently as possible, and gives bit-true agreement for the results of integer and fixed-point operations in simulation, production code, and test code.

The rest of this topic explains how to configure a hardware implementation. For details about specific parameters, see “Hardware Implementation Pane”. To see an example of **Hardware Implementation** pane capabilities, see the example model `rtwdemo_targetsettings`.

Identify the Hardware Board

Identify the hardware board that will run the code generated from your model by selecting a value for **Configuration Parameters > Hardware Implementation > Hardware board**.

If	Then Select
The menu includes the name of the hardware board that you want to use	The name of that hardware board. If you select a hardware board name, parameters for that board appear.
The name of the name of the hardware board that you want to use is missing from the menu	Get Hardware Support Packages . That value invokes the Support Package Installer. Install the target support package of interest. After you install the support package, the menu includes relevant hardware board names.
The system target file specified for the model is <code>ert.tlc</code> , <code>realtime.tlc</code> , or <code>autosar.tlc</code> .	None. No hardware board is specified for the hardware implementation.
The system target file specified for the model is a value other than <code>ert.tlc</code> , <code>realtime.tlc</code> , or <code>autosar.tlc</code> .	Determine by Code Generation system target file . The code generator uses the specified system target file to determine the hardware implementation.

The Hardware Implementation pane identifies the system target file selected on the **Code Generation** pane.

To configure test target hardware, use the Configuration Parameters **All Parameters** tab. Set `ProdEqTarget` to `off`, which enables parameters for configuring test hardware details.

Identify the Device Vendor

Use the **Device vendor** parameter to identify the vendor of the microprocessor of the target hardware device. Your selection determines the available microprocessors in the **Device type** menu. If the desired vendor name does not appear, select **Custom Processor**. Then, use the **Device type** parameter to specify the microprocessor.

- For complete lists of **Device vendor** and **Device type** values, see “Device vendor” and “Device type” in the Simulink reference documentation.
- To add **Device vendor** and **Device type** values to the default set that is displayed on the **Hardware Implementation** pane, see “Register Additional Device Vendor and Device Type Values” on page 10-5.

Identify the Device Type

Use the **Device type** parameter to identify the microprocessor name among the supported devices listed for your **Device vendor** selection. If the desired microprocessor does not appear in the menu, change **Device vendor** to **Custom Processor**.

If no available device type matches, select **Custom Processor**. Then specify device details for your custom device.

If you select a device type for which the target file specifies default hardware properties, the properties appear as initial values. You cannot change the value of parameters with only one possible selection. Parameters that have more than one possible value provide a menu. Select values for your hardware.

Register Additional Device Vendor and Device Type Values

To add **Device vendor** and **Device type** values to the default set that is displayed on the **Hardware Implementation** pane, you can use a hardware device registration API provided by the code generator.

To use this API, you create an `sl_customization.m` file, located in your MATLAB path, that invokes the `registerTargetInfo` function and fills in a hardware device registry entry with device information. The device information is registered with Simulink software for each subsequent Simulink session. (To register your device information without restarting MATLAB, issue the MATLAB command `sl_refresh_customizations`.)

For example, the following `sl_customization.m` file adds device vendor `MyDevVendor` and device type `MyDevType` to the Simulink device lists.

```
function sl_customization(cm)
    cm.registerTargetInfo(@loc_register_device);
end

function thisDev = loc_register_device
    thisDev = RTW.HWDeviceRegistry;
    thisDev.Vendor = 'MyDevVendor';
    thisDev.Type = 'MyDevType';
    thisDev.Alias = {};
    thisDev.Platform = {'Prod', 'Target'};
    thisDev.setWordSizes([8 16 32 32 32]);
    thisDev.LargestAtomicInteger = 'Char';
    thisDev.LargestAtomicFloat = 'None';
    thisDev.Endianess = 'Unspecified';
    thisDev.IntDivRoundTo = 'Undefined';
    thisDev.ShiftRightIntArith = true;
    thisDev.setEnabled({'IntDivRoundTo'});
end
```

Once registered, you can select the device in the **Hardware Implementation** pane.

To register multiple devices, specify an array of `RTW.HWDeviceRegistry` objects in your `sl_customization.m` file. For example,

```
function sl_customization(cm)
    cm.registerTargetInfo(@loc_register_device);
end

function thisDev = loc_register_device

    thisDev(1) = RTW.HWDeviceRegistry;
    thisDev(1).Vendor = 'MyDevVendor';
    thisDev(1).Type = 'MyDevType1';
    ...

    thisDev(4) = RTW.HWDeviceRegistry;
    thisDev(4).Vendor = 'MyDevVendor';
    thisDev(4).Type = 'MyDevType4';
    ...

end
```


The following table lists the `RTW.HWDeviceRegistry` properties that you can specify in the `registerTargetInfo` function call in your `sl_customization.m` file.

Property	Description
Vendor	Character vector specifying the Device vendor value for your hardware device.
Type	Character vector specifying the Device type value for your hardware device.
Alias	Cell array of character vectors specifying aliases or legacy names that users might specify that should resolve to this device. Specify each alias or legacy name in the format <code>'Vendor->Type'</code> . (Embedded Coder software provides the utility functions <code>RTW.isHWDeviceTypeEq</code> and <code>RTW.resolveHWDeviceType</code> for detecting and resolving alias values or legacy values when testing user-specified values for the target device type.)
Platform	Cell array of enumerated character vector values specifying whether this device should be listed in the Production hardware subpane (<code>{'Prod'}</code>), the Test hardware subpane (<code>{'Target'}</code>), or both (<code>{'Prod', 'Target'}</code>).
setWordSizes	Array of integer sizes to associate with the Number of bits parameters char , short , int , long , and native word size , respectively.
LargestAtomicInteger	Character vector specifying an enumerated value for the Largest atomic size: integer parameter: <code>'Char'</code> , <code>'Short'</code> , <code>'Int'</code> , or <code>'Long'</code> .
LargestAtomicFloat	Character vector specifying an enumerated value for the Largest atomic size: floating-point parameter: <code>'Float'</code> , <code>'Double'</code> , or <code>'None'</code> .
Endianness	Character vector specifying an enumerated value for the Byte ordering parameter: <code>'Unspecified'</code> , <code>'Little'</code> for little Endian, or <code>'Big'</code> for big Endian.
IntDivRoundTo	Character vector specifying an enumerated value for the Signed integer division rounds to parameter: <code>'Zero'</code> , <code>'Floor'</code> , or <code>'Undefined'</code> .

Property	Description
ShiftRightIntArith	Boolean value specifying whether your compiler implements a signed integer right shift as an arithmetic right shift (true) or not (false).
setEnabled	Cell array of character vectors specifying which device properties should be enabled (modifiable) in the Hardware Implementation pane when this device type is selected. In addition to the 'Endianness', 'IntDivRoundTo', and 'ShiftRightIntArith' properties listed above, you can enable individual Number of bits parameters using the property names 'BitPerChar', 'BitPerShort', 'BitPerInt', 'BitPerLong', and 'NativeWordSize'.

Set Bit Lengths for Device Data Types

The **Number of bits** parameters describe the **native word size** of the microprocessor, and the bit lengths of **char**, **short**, **int**, and **long** data. For code generation to succeed:

- The bit lengths must be such that **char** <= **short** <= **int** <= **long**.
- Bit lengths must be multiples of 8, with a maximum of 32.
- The bit length for **long** data must not be less than 32.

Integer type names are defined in the file `rtwtypes.h`. The values that you provide must be consistent with the word sizes as defined in the compiler's `limits.h` header file. The following table lists code generator integer type names and maps them to the corresponding Simulink names.

Code Generator Integer Type	Simulink Integer Type
<code>boolean_T</code>	<code>boolean</code>
<code>int8_T</code>	<code>int8</code>
<code>uint8_T</code>	<code>uint8</code>
<code>int16_T</code>	<code>int16</code>
<code>uint16_T</code>	<code>uint16</code>
<code>int32_T</code>	<code>int32</code>
<code>uint32_T</code>	<code>uint32</code>

If no ANSI[®] C type with a matching word size is available, but a larger ANSI C type is available, the code generator uses the larger type for `int8_T`, `uint8_T`, `int16_T`, `uint16_T`, `int32_T`, and `uint32_T`. When the code generator uses a larger type, the resulting logged values (for example, using MAT-file logging) can have different data types than logged values used for simulation.

An application can use integer data of length from 1 (unsigned) or 2 (signed) bits up to 32 bits. If the integer length matches the length of an available type, the code generator uses that type. If a matching type is not available, the code generator uses the smallest available type that can hold the data, generating code that does not use unnecessary higher-order bits. For example, on a target that provided 8-bit, 16-bit, and 32-bit integers, a signal specified as 24 bits would be implemented as an `int32_T` or `uint32_T`.

Note: During code generation, the software checks the compatibility of model data types with the data types that you specify for production hardware:

- If none of the lengths specified for production hardware integers is 32 bits, the software generates an error like the following:

Error in embedded hardware settings on Hardware Implementation pane of Configuration Parameters dialog box: at least one of "char", "short", "int" or "long" must have a value of 32

- If the lengths of data types used by the model are smaller than the available production hardware integer lengths, the software generates a warning like the following:

Warning: The data type "int16" uses a word size that is not available on the intended target. Fixed-point signals using this data type will be put inside a larger word or multi words. When used, extra software will be generated to force this larger word or multi words to emulate a smaller word. This emulation is helpful when your prototype target and your final production target are not the same. If the smaller word size does NOT exist on the final production target, then consider increasing the word size to one that is supported.

Code that uses emulated integer data is not maximally efficient, but can be useful during application development for emulating integer lengths that are available only on production hardware. The use of emulation does not affect the results of execution.

Set Byte Ordering Used By Device

The **Byte ordering** parameter specifies whether the target hardware uses **Big Endian** (most significant byte first) or **Little Endian** (least significant byte first) byte ordering. If left as **Unspecified**, the code generator produces code that determines the endianness of the target; this is the least efficient setting.

Set Quotient Rounding Behavior for Signed Integer Division

ANSI C does not completely define the quotient rounding technique to be used when dividing one signed integer by another, so the behavior is implementation-dependent. If both integers are positive, or both are negative, the quotient must round down. If either integer is positive and the other is negative, the quotient can round up or down.

The **Signed integer division rounds to** parameter tells the code generator how the compiler rounds the result of signed integer division. Providing this information does not change the operation of the compiler. It only describes that behavior to the code generator, which uses the information to optimize code generated for signed integer division. The parameter values are:

- **Zero** — If the quotient is between two integers, the compiler chooses the integer that is closer to zero as the result.
- **Floor** — If the quotient is between two integers, the compiler chooses the integer that is closer to negative infinity.
- **Undefined** — Choose this value if neither **Zero** nor **Floor** describes the compiler's behavior, or if that behavior is unknown.

The following table illustrates the compiler behavior that corresponds to each of these three values:

N	D	Ideal N/D	Zero	Floor	Undefined
33	4	8.25	8	8	8
-33	4	-8.25	-8	-9	-8 or -9
33	-4	-8.25	-8	-9	-8 or -9
-33	-4	8.25	8	8	8 or 9

Note: Select **Undefined** only as a last resort. When the code generator does not know the signed integer division rounding behavior of the compiler, it generates extra code.

The compiler's implementation for signed integer division rounding can be obtained from the compiler documentation, or by experiment if documentation is not available.

Set Arithmetic Right Shift Behavior for Signed Integers

ANSI C does not define the behavior of right shifts on negative integers, so the behavior is implementation-dependent. The **Shift right on a signed integer as arithmetic shift** option tells the code generator how the compiler implements right shifts on negative integers. Providing this information does not change the operation of the compiler. It only describes that behavior to the code generator, which uses the information to optimize the code generated for arithmetic right shifts.

Select the option if the C compiler implements a signed integer right shift as an arithmetic right shift, and clear the option otherwise. An arithmetic right shift fills bits vacated by the right shift with the value of the most significant bit, which indicates the sign of the number in two's-complement notation. The option is selected by default. If your compiler handles right shifts as arithmetic shifts, this is the preferred setting.

- When you select the option, the code generator produces efficient code whenever the Simulink model performs arithmetic shifts on signed integers.
- When the option is cleared, the code generator produces fully portable but less efficient code to implement right arithmetic shifts.

The compiler's implementation for arithmetic right shifts can be obtained from the compiler documentation, or by experiment if documentation is not available.

Update Release 14 Hardware Configuration

If your model was created before Release 14 and has not been updated, the **Configuration Parameters > All Parameters > Configure current execution hardware device** parameter (TargetUnknown) value is 'on' by default.

To update your model, uncheck the box for **Configuration Parameters > All Parameters > Configure current execution hardware device** or type the following at the MATLAB command line:

```
cs = getActiveConfigSet('your_model_name');  
set_param(cs, 'TargetUnknown', 'off');
```

After setting the parameter value to 'off', this update to your model:

- Enables operation of the **Test Hardware is the same as production hardware** parameter (ProdEqTarget) and sets this parameter value to 'on'
- Copies the **Production device vendor and type** parameter (ProdHWDeviceType) value to the **Test device vendor and type** parameter (TargetHWDeviceType)

Uncheck the box for **Configuration Parameters > All Parameters > Test Hardware is the same as production hardware** (only if your production and test hardware *are different*), set the parameters in **Configuration Parameters > All Parameters > Hardware implementation** to match your production and test systems, and save the model to complete the update.

Configure Production Hardware Characteristics

“Configure Target Hardware” on page 10-3 explains how to configure target hardware for a model. This topic describes considerations that apply only to production target hardware. When you configure production hardware, keep the following in mind:

- Code generation targets can have word sizes and other hardware characteristics that differ from the MATLAB host. Furthermore, code can be prototyped on hardware that is different from the deployment target or the MATLAB host. The code generator takes these differences into account when producing code.
- The Simulink product uses some of the information in the production target hardware configuration. That information enables simulations without code generation to give the same results as executing generated code. For example, the results can detect error conditions that could arise on the target hardware, such as hardware overflow.
- The code generator produces code that has bit-true agreement with Simulink results for integer and fixed-point operations. Generated code that emulates unavailable data lengths runs less efficiently than it would without emulation, but the emulation does not affect bit-true agreement with Simulink for integer and fixed-point results.
- If you change targets during application development, reconfigure the hardware implementation parameters for the new target before generating or regenerating code. Bit-true agreement might not be achieved for results of integer and fixed-point operations in simulation, production code, and test code when code executes on hardware for which it was not generated.
- Use the **Integer rounding mode** parameter on model blocks to simulate the rounding behavior of the C compiler that you intend to use to compile code generated from the model. This setting appears on the **Signal Attributes** pane of the parameter dialog boxes of blocks that can perform signed integer arithmetic, such as the **Product** and **n-D Lookup Table** blocks.
- For most blocks, the value of **Integer rounding mode** completely defines rounding behavior. For blocks that support fixed-point data and the simplest rounding mode, the value of **Signed integer division rounds to** also affects rounding. For details, see “Precision”.
- When models contain Model blocks, configure models that they reference to use identical hardware settings.

Configure Test Hardware Characteristics

“Configure Target Hardware” on page 10-3 explains how to configure target hardware for a model. This topic describes considerations that apply only to production target hardware.

By default, the test target hardware configuration is the same as the configuration for the production target hardware. Code generated under this configuration is suitable for production use, or for testing in an environment identical to the production environment.

To generate code that runs on a test target hardware for test purposes, but behaves as if it were running on production target hardware in a production application, adjust the settings of parameters for test target hardware.

To configure test target hardware:

- 1** Open the Configuration Parameters **All Parameters** tab.
- 2** Enable parameters for configuring test target hardware by setting **ProdEqTarget** to off.
- 3** Use the test hardware (**Target***) parameters to adjust the device type details.

If you have specified a system target file, and the target file specifies a default microprocessor and its hardware properties, the default and properties appear as initial values.

Parameters with only one possible value cannot be changed. If you modify hardware properties, check carefully that their values are consistent with the system target file. Otherwise, the generated code might fail to compile or execute, or might execute but give incorrect results.

Control the Location for Generated Files

By default, the files generated by Simulink diagram updates and model builds are placed in a build folder, the root of which is the current working folder (pwd). If you are doing model builds, which potentially generate files for simulation targets and code generation targets, artifacts used for simulation and code generation files coexist in subfolders within that build folder. However, in some situations, you might want the generated files to go to a root folder outside the current working folder. For example,

- You need to keep generated files separate from the models and other source materials used to generate them.
- You want to reuse or share previously-built simulation targets without having to set the current working folder back to a previous working folder.

You might also want to separate generated simulation artifacts from generated production code.

To allow you to control the output locations for the files generated by diagram updates and model builds, the software allows you to separately specify the following build folders:

- *Simulation cache folder* — root folder in which to place build artifacts used for simulation
- *Code generation folder* — root folder in which to place code generation files

For specifying the folder locations, the software provides

- MATLAB session parameters `CacheFolder` and `CodeGenFolder`
- Simulink preferences **Simulation cache folder** and **Code generation folder**, which, if specified, provide the initial defaults for the MATLAB session parameters
- Function `Simulink.fileGenControl` for directly manipulating the MATLAB session parameters, for example, overriding or restoring the initial default values for the current session

For more information about setting up a simulation cache folder, see “Control Generated Files Location Used for Simulation” on page 10-17.

For more information about setting up a code generation folder, see “Control the Location for Code Generation Files” on page 10-19.

For more information about directly manipulating the MATLAB session parameters `CacheFolder` and `CodeGenFolder`, see “Override Build Folder Settings for Current Session” on page 10-21.

Control Generated Files Location Used for Simulation

By default, the files generated by Simulink diagram updates are placed in a build folder, the root of which is the current working folder (pwd). However, in some situations, you might want the generated files to go to a root folder outside the current working folder. For example,

- You need to keep generated files separate from the models and other source materials used to generate them.
- You want to reuse or share previously-built simulation targets without having to set the current working folder back to a previous working folder.

The Simulink preference **Simulation cache folder** provides control over the output location for files generated by Simulink diagram updates. The preference appears in the **General** pane, under **File generation control**. To specify the root folder location for files generated by Simulink diagram updates, set the preference value by entering or browsing to a folder path.

The folder path that you specify provides the initial default for the MATLAB session parameter `CacheFolder`. When you initiate a Simulink diagram update, files generated are placed in a build folder at the root location specified by `CacheFolder` (if any), rather than in the current working folder (pwd).

For example, using a 32-bit Windows host platform, if you set the **Simulation cache folder** to 'C:\Work\mymodelsimcache' and then simulate the example model `rtwdemo_capi`, files are generated into the specified folder.

As an alternative to using the Simulink preferences GUI to set **Simulation cache folder**, you also can get and set the preference value from the command line using `get_param` and `set_param`. For example,

```
>> get_param(0, 'CacheFolder')  
  
ans =  
  
    ''  
  
>> set_param(0, 'CacheFolder', fullfile('C:', 'Work', 'mymodelsimcache'))  
>> get_param(0, 'CacheFolder')  
  
ans =  
  
C:\Work\mymodelsimcache
```

Also, you can choose to override the **Simulation cache folder** preference value for the current MATLAB session.

Control the Location for Code Generation Files

By default, the files generated by Simulink model builds are placed in a build folder, the root of which is the current working folder (pwd). Model builds potentially generate files for simulation targets as well as code generation targets, and the resulting build folder contains both artifacts used for simulation and code generation files. However, in some situations, you might want the generated files to go to one or more root folders outside the current working folder. For example,

- You need to keep generated files separate from the models and other source materials used to generate them.
- You want to separate generated production code from generated simulation artifacts.

The Simulink preference **Code generation folder** provides control over the output location for files generated by model builds for code generation targets. The preference appears in the **General** pane, under **File generation control**. To specify the root folder location for code generation files generated by model builds, set the preference value by entering or browsing to a folder path.

The folder path that you specify provides the initial default for the MATLAB session parameter `CodeGenFolder`. When you initiate a Simulink model build, code generation files generated are placed in a build folder at the root location specified by `CodeGenFolder` (if any), rather than in the current working folder (pwd).

For example, using a 32-bit Windows host platform, if you set the **Code generation folder** to `'C:\test\mymodelgencode'` and then build the example model `rtwdemo_capi`, files are generated into the specified folder.

As an alternative to using the Simulink preferences GUI to set **Code generation folder**, you also can get and set the preference value from the command line using `get_param` and `set_param`. For example,

```
>> get_param(0, 'CodeGenFolder')
ans =
    ''

>> set_param(0, 'CodeGenFolder', fullfile('C:', 'test', 'mymodelgencode'))
>> get_param(0, 'CodeGenFolder')
ans =
C:\test\mymodelgencode
```

Also, you can choose to override the **Code generation folder** preference value for the current MATLAB session. For more information, see “Override Build Folder Settings for Current Session” on page 10-21.

Override Build Folder Settings for Current Session

The Simulink preferences **Simulation cache folder** and **Code generation folder** provide the initial defaults for the MATLAB session parameters `CacheFolder` and `CodeGenFolder`, which determine where files generated by Simulink diagram updates and model builds are placed. However, you can override these build folder settings during the current MATLAB session, using the `Simulink.fileGenControl` function. This function allows you to directly manipulate the MATLAB session parameters, for example, overriding or restoring the initial default values. The values you set using `Simulink.fileGenControl` expire at the end of the current MATLAB session. For more information and detailed examples, see the `Simulink.fileGenControl` function reference page.

Model Protection in Simulink Coder

- “Protect a Referenced Model” on page 11-2
- “Harness Model” on page 11-4
- “Protected Model Report” on page 11-5
- “Code Generation Support in a Protected Model” on page 11-6
- “Protected Model File” on page 11-8
- “Create a Protected Model” on page 11-10
- “Protected Model Creation Settings” on page 11-15
- “Create a Protected Model with Multiple Targets” on page 11-18
- “Use a Protected Model with Multiple Targets” on page 11-19
- “Test the Protected Model” on page 11-20
- “Save Base Workspace Definitions” on page 11-22
- “Package a Protected Model” on page 11-23
- “Specify Custom Obfuscator for Protected Model” on page 11-24
- “Define Callbacks for Protected Model” on page 11-26

Protect a Referenced Model

Protect a model when you want to share a model with a third party without revealing intellectual property. Protecting a model does not use encryption technology unless you use the optional password protection available for read-only view, simulation, and code generation. If you choose password protection for one of these options, the software protects the supporting files using AES-256 encryption.

When you create a protected model:

- By default, Simulink creates and stores a protected version of the referenced model in the current working folder. The protected model has the same name as the source model, with a `.slxp` extension.
- The original Model block does not change. However, if the Model block parameter **Model name** does not specify an extension, a protected model, `.slxp`, takes precedence over a model file, `.slx`.
- You can optionally create a harness model which includes the protected model. A shield icon appears in the lower-left corner of the protected model block in the harness model. For more information, see “Harness Model” on page 11-4.
- You can optionally include generated code with the protected model so that a third party can generate code for a model that contains the protected model. For more information, see “Code Generation Support in a Protected Model” on page 11-6.
- If the Model block uses variants, only the active variant is protected. For more information, see “Set up Model Variants”.
- If the model defines callbacks, the model protection process does not preserve these callbacks. For more information on creating callbacks for use with a protected model, see “Define Callbacks for Protected Model” on page 11-26.
- If you rename a protected model, or change its suffix, the model is unusable until you restore its original name and suffix. You cannot change a protected model file internally because such changes make the file unusable.

Create a protected model using one of the following options.

- The Model block context menu. For more information, see “Create a Protected Model” on page 11-10
- The `Simulink.ModelReference.protect` function.
- The Simulink Editor menu bar. Select **File > Export Model To > Protected Model** to create a protected model from the current model.

Requirements for Protecting a Model

When you create a protected model from a referenced model, the referenced model must meet all requirements listed in “Model Referencing Limitations”, as well as these requirements:

- You must have a Simulink Coder license to create a protected model.
- A model that you protect must be available on the MATLAB path and not have unsaved changes.
- A model that you protect cannot reference a protected model directly or indirectly.
- A model that you protect cannot use a non-inlined S-function directly or indirectly.
- To use a protected model that requires passwords across platforms, before you create the protected model, set the MATLAB character set encoding to 'US-ASCII'. For more information, see `slCharacterEncoding`.

Model protection has certain limitations, as listed in “Limitations on All Model Referencing” and “Limitations on Accelerator Mode Referenced Models”.

Harness Model

You can create a harness model for the generated protected model. The harness model opens as a new, untitled model that contains only a Model block that references the protected model. This Model block:

- Specifies the Model block parameter, **Model name**, as the name of the protected model.
- Has a shield icon in the lower-left corner.
- Has the same number of input and output ports as the protected model.
- Defines model reference arguments that the protected model uses, but does not provide values.

To create a harness model, see “Create a Protected Model” on page 11-10. You can use a harness model to test your protected model. For more information, see “Test the Protected Model” on page 11-20. You can also copy the Model block in your harness model to another model, where it is an interface to the protected model.

Protected Model Report

You can generate a protected model report when you create the protected model. The report is included as part of the protected model. The report has:

- A **Summary**, including the following tables:
 - **Environment**, providing the Simulink version and platform used to create the protected model.
 - **Supported functionality**, reporting On, Off, or On with password protection for each possible functionality that the protected model supports.
 - **Licenses**, listing licenses required to run the protected model.
- An **Interface Report**, including model interface information such as input and output specifications, exported function information, interface parameters, and data stores.

When you create the protected model from the Simulink Editor, the protected model report is generated. To generate a report when using the `Simulink.ModelReference.protect` function, set the 'Report' option to `true`.

If you configure your protected model for multiple targets, the **Summary** includes a list of supported targets in the **Supported functionality** table. When you build a model that references a protected model with multiple targets, the protected model code generation report represents the currently configured target.

To view the protected model report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the `report` option.

Code Generation Support in a Protected Model

You can create a protected model that supports code generation. When a protected model includes generated code, a third party can generate code for a model that includes the protected model. If you choose to obfuscate the code, the code is obfuscated before compilation. The protected model file contains only obfuscated headers and binaries. Source code, such as `.c` and `.cpp`, is not present in the protected model file, although the headers are documented in the protected model report. For more information, see “Protected Model File” on page 11-8 and “Protected Model Report” on page 11-5.

In the Create Protected Model dialog box, select the **Use generated code** check box. The appearance of the generated code is determined by the **Content type** list. To enable obfuscated code, select **Obfuscated source code** from the list. For an example on including code generation support, see “Create a Protected Model” on page 11-10.

Protected Model Requirements to Support Code Generation

Contents and configuration of a model might prevent code generation support of the protected model. Interaction between the parent model and the protected model might also prevent code generation.

- Code generation for the protected model is only supported for Normal, Accelerator, Software-in-the-Loop (SIL), and Processor-in-the-Loop (PIL) modes and a single target. Both GRT and ERT targets cannot be supported by the same protected model.
- Source code comments in the **Code Generation > Comments** pane are ignored. Obfuscation of the generated code removes comments because comments might reveal intellectual property.
- Custom code specified in the **Code Generation > Custom Code** pane is obfuscated, but identifiers are not.
- Code generation of a model that includes a protected model causes an error, if:
 - Their interfaces do not match.
 - There are incompatible parameters.
 - A protected model and another model share the same name in the same model reference hierarchy.
- Selecting the **Code Generation > Verification > Measure function execution time** check box is incompatible with model protection. If you have this option selected

when you protect your model, the software turns the parameter off and displays a warning.

Protected Model File

A protected model file (.slxp) consists of the model itself and supporting files, depending on the options that you selected when you created the protected model.

If you created a protected model for simulation only and the referencing model is in Normal mode, after simulation, the *model.mexext* file is placed in the build folder.

If you created a protected model for simulation only and the referencing model is in Accelerator or Rapid Accelerator mode, after simulation, the following files are unpacked:

- `slprj/sim/model/*.h`
- `slprj/sim/model/modellib.a` (or `modellib.lib`)
- `slprj/sim/model/tmwinternal/*`
- `slprj/sim/_sharedutils/*`

For the protected model report, these additional files are unpacked (but not in the build folder):

- `slprj/sim/model/html/*`
- `slprj/sim/model/buildinfo.mat`

If you opted to include code generation support when you created the protected model, after building your model the following files are unpacked (in addition to the preceding files):

- `slprj/sim/model/*.h`
- `slprj/sim/model/modellib.a` (or `modellib.lib`)
- `slprj/sim/model/tmwinternal/*`
- `slprj/sim/_sharedutils/*`
- `slprj/target/model/*.h`
- `slprj/target/model/model_rtwlib.a` (or `model_rtwlib.lib`)
- `slprj/target/model/buildinfo.mat`
- `slprj/target/model/codeinfo.mat`
- `slprj/target/_sharedutils/*`
- `slprj/target/model/tmwinternal/*`

With an Embedded Coder license, you can specify a `Top model` code interface. In this case, if you opted to include code generation support when you created the protected model, after building your model the following files are unpacked:

- `slprj/sim/model/*.h`
- `slprj/sim/model/modellib.a` (or `modellib.lib`)
- `slprj/sim/model/tmwinternal/*`
- `slprj/sim/_sharedutils/*`
- `model_target_rtw/*.h`
- `model_target_rtw/*.objExt`
- `model_target_rtw/buildinfo.mat`
- `model_target_rtw/codeinfo.mat`
- `slprj/target/_sharedutils/*`
- `slprj/target/model/tmwinternal/*`

For the protected model report, after building your model these files are unpacked (in addition to the preceding files):

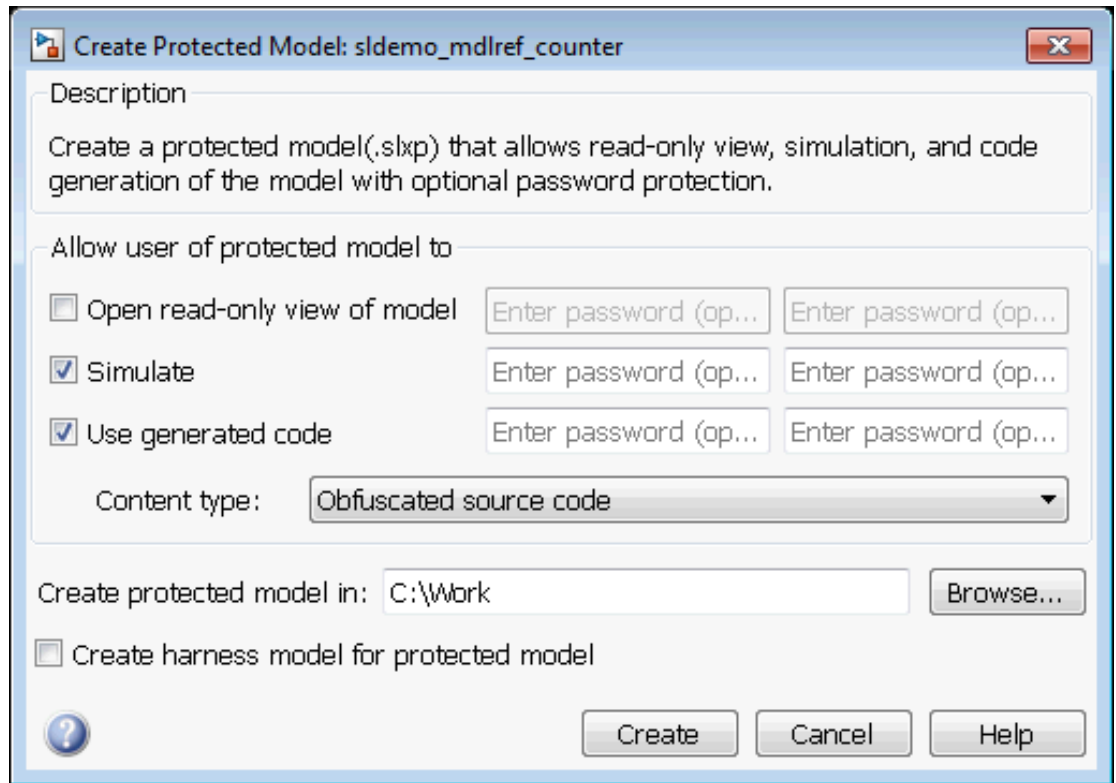
- `slprj/target/model/html/*`
- `slprj/target/model/buildinfo.mat`
- `slprj/target/_sharedutils/html/*`

Note: The `slprj/sim/model/*` files are deleted after they are used.

Create a Protected Model

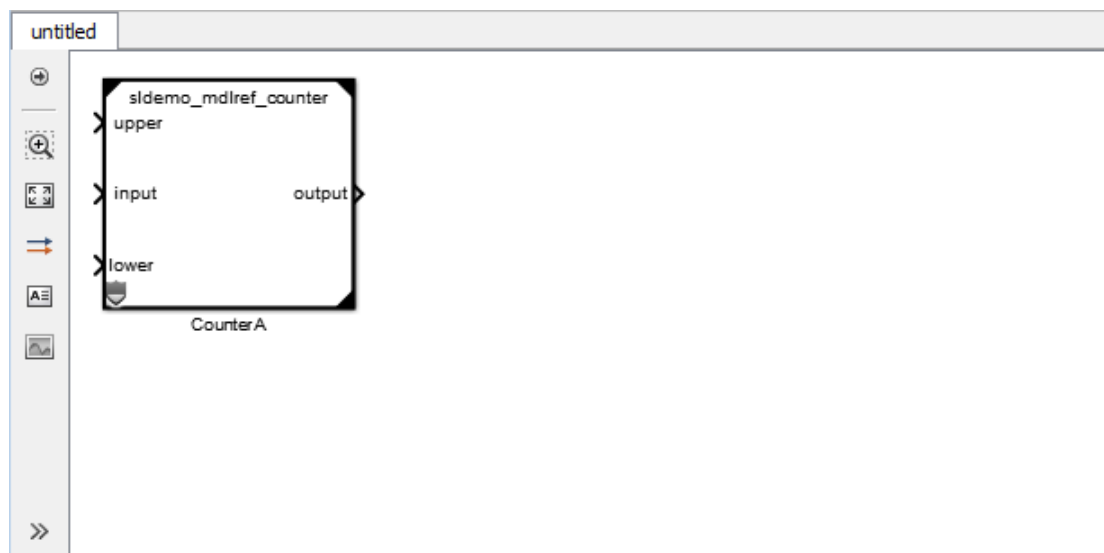
This example shows how to create a protected model for read-only viewing, simulation, or code generation.

- 1** Open your model. For this example, `sldemo_md1ref_basic` is used as a demonstration.
- 2** In the Simulink Editor, right-click the model block that references the model for which you want to generate protected model code. In this example, right-click CounterA.
- 3** From the context menu, select **Block Parameters (ModelReference)**.
- 4** In the Block Parameters dialog box, in the **Model name** field, specify the extension for the model, `.slx`. When both the model and the protected model exist in the same folder, `.slxp` takes precedence over `.slx`. In the **Model name** field, if you do not specify an extension, then the original model block in the model becomes protected.
- 5** Click **Apply** and **OK**.
- 6** Right-click the model block. From the context menu, select **Subsystem & Model Reference > Create Protected Model for Selected Model Block**.

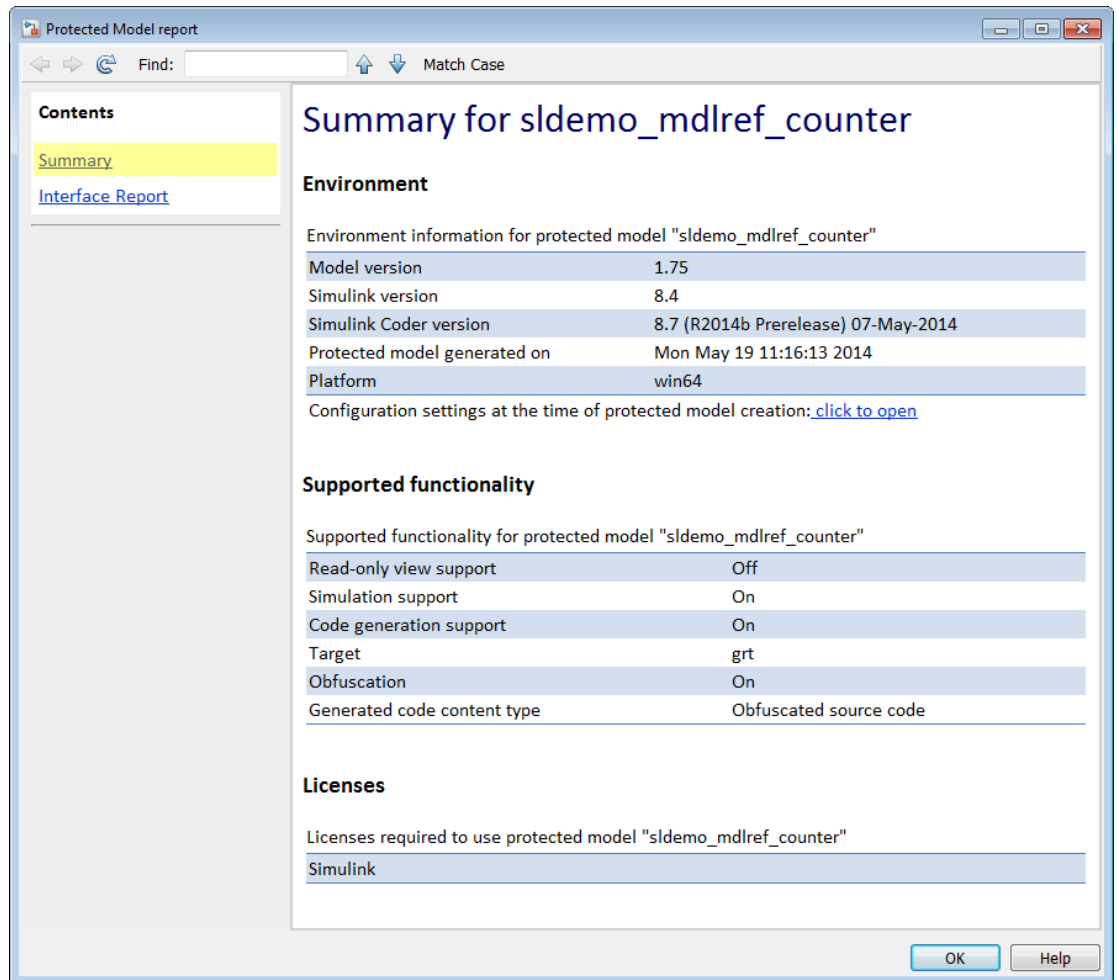


- 7 In the Create Protected Model dialog box, select the **Simulate** and **Use generated code** check boxes. If you want to password-protect the functionality of the protected model, enter a password with a minimum of four characters. Each functionality can have a unique password.
- 8 If you have an Embedded Coder license and specify an ERT based system target file (for example, `ert.tlc`) for the model, the **Code interface** field is visible. From the **Code interface** drop-down list, select one of the following options:
 - **Model reference** — Specifies code access through the model reference code interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block SIL/PIL simulations with the protected model.

- **Top model** — Specifies code access through the standalone interface. Users of the protected model can run Model block SIL/PIL simulations with the protected model.
- 9 From the **Content type** list, select **Obfuscated source code** to conceal the source code purpose and logic of the protected model.
 - 10 In the **Create protected model in** field, specify the folder path for the protected model. The default value is the current working folder.
 - 11 To create a harness model for the protected model, select the **Create harness model for protected model** check box.
 - 12 Click **Create**. An untitled harness model opens. It contains a model block, which refers to the protected model `sldemo_mdhref_counter.slxp`. The **Simulation mode** for the Model block is set to **Accelerator**. You cannot change the mode.



- 13 To view the protected model report, right-click the protected-model badge icon on the CounterA block and select **Display Report**.



Related Examples

- “Test the Protected Model” on page 11-20
- “Package a Protected Model” on page 11-23
- “Configure and Run SIL Simulation”

More About

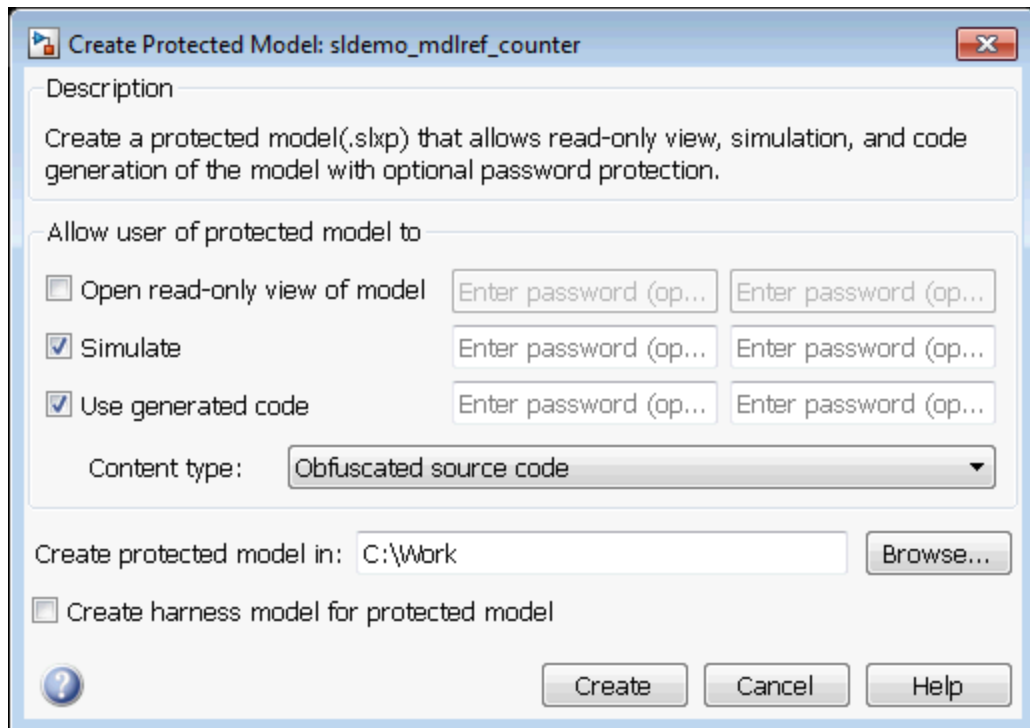
- “Code Generation Support in a Protected Model” on page 11-6
- “Protected Model Creation Settings” on page 11-15
- “Code Interfaces for SIL and PIL”

Protected Model Creation Settings

When you create a protected model, in the Create Protected Model dialog box, you can select which settings you want configured. The settings provide certain functionality permissions when using a protected model. The functionality choices are:

- Read-only viewing
- Simulation
- Code Generation

Password-protection is optional. You must have a minimum of four characters.



Open Read-Only View of Model

If you want to share a view-only version of your model, this option will allow someone using the protected model to open a Web view of the model. The contents and block parameters are viewable in the model Web view.

Simulate

The **Simulate** check box allows someone to simulate a protected model. When you select this check box, the Web view is not inherited. To enable the Web view with simulation functionality, select the **Open Read-Only View of Model** check box. The **Simulate** functionality:

- Enables protected model Simulation Report.
- Sets Mode to Accelerator. You can run Normal Mode and Accelerator simulations.
- Displays only binaries and headers.
- Enables code obfuscation.

Use Generated Code

The **Use generated code** check box allows simulation and code generation for a protected model. To generate code, the **Simulate** check box must also be selected. This functionality:

- Enables protected model Simulation Report and Code Generation Report.
- Sets Mode to enable code generation.
- Enables support for simulation.
- Supports the Model block if you have an Embedded Coder license and specify an ERT system target file (`ert.tlc`) for the model. From the **Code interface** drop-down list, select one of the following options:
 - **Model reference** — Specifies the model reference code interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block SIL/PIL simulations with the protected model.
 - **Top model** — Specifies code access through the standalone interface. Users of the protected model can run Model block SIL/PIL simulations with the protected model.

- Determines the appearance of the generated code by the **Content type** list. The options are:
 - **Binaries**
 - **Obfuscated source code** (default)
 - **Readable source code**, which also includes readable code comments

Create a Protected Model with Multiple Targets

You can create a protected model that supports multiple code generation targets. This example shows how to use command-line functions to create a protected model that supports code generation for GRT and ERT targets.

- 1 Load a model and save a local copy. This model is configured for a GRT target.

```
sldemo_mdhref_counter
save_system('sldemo_mdhref_counter', 'mdlref_counter.slx');
```

- 2 Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter', 'password');
```

- 3 Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter', 'Mode', ...
'CodeGeneration', 'Modifiable', true, 'Report', true);
```

- 4 Get a list of targets that the protected model supports.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')

st =

    'grt'    'sim'
```

- 5 Configure the unprotected model to support an ERT target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

- 6 Add support to the protected model for the ERT target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

- 7 Verify that the list of supported targets now includes the ERT target.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')

st =

    'ert'    'grt'    'sim'
```

Use a Protected Model with Multiple Targets

When using a protected model with multiple targets, prepare your model for code generation.

- 1 Get a list of the targets that the protected model supports using the `Simulink.ProtectedModel.getSupportedTargets` function.

You can also get this information from the protected model report. To view the report, on the protected model block, right-click the badge icon. Select **Display Report**. The **Summary** lists the supported targets.

- 2 Get the configuration set for your chosen target using the `Simulink.ProtectedModel.getConfigSet` function. You can use the configuration set to verify that the protected model interface is compatible with the parent model.
- 3 Generate code. The build process selects the correct target.

Test the Protected Model

To test a protected model that you created, you use the generated harness model and the procedure described in “Use Protected Model in Simulation”.

You can also compare the output of the protected model to the output of the original model. Because you are the supplier, both the original and the protected model might exist on the MATLAB path. In the original model, if the Model block **Model name** parameter names the model without providing a suffix, the protected model takes precedence over the unprotected model. If you need to override this default when testing the output, in the Model block **Model name** parameter, specify the file name with the extension of the unprotected model, `.slx`.

To compare the unprotected and protected versions of a Model block, use the Simulation Data Inspector. This example uses `sldemo_mdhref_basic` and the protected model, `sldemo_mdhref_counter.slxp`, which is created in “Create a Protected Model” on page 11-10.

- 1 If it is not already open, open `sldemo_mdhref_basic`.
- 2 Enable logging for the output signal of the Model block, CounterA. In the Configuration Parameters dialog box, in the **Data Import/Export** pane, select the **Signal logging** parameter. Click **Apply** and **OK**.
- 3 Right-click the output signal. From the context menu, select **Properties**. In the Signal Properties dialog box, select **Log signal data**. Click **Apply** and **OK**. For more information, see “Export Signal Data Using Signal Logging”.
- 4 Right-click the CounterA block. From the context menu, select **Block Parameters (ModelReference)**. In the Block Parameters dialog box, specify the **Model name** parameter with the name of the unprotected model and the extension, `sldemo_mdhref_counter.slx`. Click **Apply** and **OK**. Repeat this for CounterB block and CounterC block.
- 5 In the Simulink Editor, click the **Simulation Data Inspector** button arrow and select **Send Logged Workspace Data to Data Inspector** from the menu.
- 6 Simulate the model. When the simulation is complete, click the **Simulation Data Inspector** button to open the Simulation Data Inspector.
- 7 In the Simulation Data Inspector, rename the run to indicate that it is for the unprotected model.
- 8 In the Simulink Editor, right-click the CounterA block. From the context menu, select **Block Parameters (ModelReference)**. In the Block Parameters dialog

box, specify the **Model name** parameter with the name of the protected model, `sldemo_md1ref_counter.slxp`. A shield icon appears on the Model block. Repeat this for CounterB block and CounterC block.

- 9 Simulate the model, which now refers to the protected model. When the simulation is complete, a new run appears in the Simulation Data Inspector.
- 10 In the Simulation Data Inspector, rename the new run to indicate that it is for the protected model.
- 11 In the Simulation Data Inspector, click the **Compare** tab. From the **Baseline** and **Compare To** lists, select the runs from the unprotected and protected model, respectively. Click **Compare Runs** to compare the runs. For more information about comparing runs, see “Compare Signal Data from Multiple Simulations”.

Save Base Workspace Definitions

Referenced models might use object definitions or tunable parameters that are defined in the MATLAB base workspace. These variables are not saved with the model. When you protect a model, you must obtain the definitions of required base workspace entities and ship them with the model.

The following base workspace variables must be saved to a MAT-file:

- Global tunable parameter
- Global data store
- The following objects used by a signal that connects to a root-level model Inport or Outport:
 - `Simulink.Signal`
 - `Simulink.Bus`
 - `Simulink.Alias`
 - `Simulink.NumericType` that is an alias

For more information, see “Edit and Manage Workspace Variables Used by Models”.

Before executing the protected model as a part of a third-party model, the receiver of the protected model must load the MAT-file.

Package a Protected Model

In addition to the protected model file (.slxp), you might need to include additional files in the protected model package:

- Harness model file.
- Any required definitions saved in a MAT-file. For more information, see “Save Base Workspace Definitions” on page 11-22.
- Instructions on how to retrieve the files.

Some ways to deliver the protected model package are:

- Provide the .slxp file and other supporting files as separate files.
- Combine the files into a ZIP or other container file.
- Combine the files using a manifest. For more information, see “Export Files in a Manifest”.
- Provide the files in some other standard or proprietary format specified by the receiver.

Whichever approach you use to deliver a protected model, include information on how to retrieve the original files. One approach to consider is to use the Simulink Manifest Tools, as described in “Analyze Model Dependencies”.

Specify Custom Obfuscator for Protected Model

When creating a protected model, you can specify your own postprocessing function for files that the protected model creation process generates. Prior to packaging the protected model files, this function is called by the `Simulink.ModelReference.protect` function. You can use this functionality to run your own custom obfuscator on the generated files by following these steps:

- 1 Create your postprocessing function. Use this function to call your custom obfuscator. The function must be on the MATLAB path and accept a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable.
- 2 In your function, get the files and exported symbol information that your custom obfuscator requires to process the protected model files. To get the files and information, access the properties of your function input variable. The variable is a `Simulink.ModelReference.ProtectedModel.HookInfo` object with the following properties:
 - `SourceFiles`
 - `NonSourceFiles`
 - `ExportedSymbols`
- 3 Pass the protected model file information to your custom obfuscator. The following is an example of a postprocessing function for custom obfuscation:

```
function myHook(protectedModelInfo)

    % Get source file list information.
    srcFileList = protectedModelInfo.SourceFiles;
    disp('### Obfuscating...');
    for i=1:length(srcFileList)
        disp(['### Obfuscator: Processing ' srcFileList{i} '...']);
        % call to custom obfuscator
        customObfuscator(srcFileList{i});
    end
end
```

- 4 Specify your postprocessing function when creating the protected model:

```
Simulink.ModelReference.protect('myModel', 'Mode', 'CodeGeneration', ...
    'CustomPostProcessingHook', ...
    @(protectedModelInfo)myHook(protectedModelInfo))
```


The creator of the protected model also has the option of enabling obfuscation of simulation target code and generated code through the 'ObfuscateCode' option of the `Simulink.ModelReference.protect` function. Your custom obfuscator runs only on the generated code and not on the simulation target code. If both obfuscators are in use, the custom obfuscator is the last to run on the generated code before the files are packaged.

Define Callbacks for Protected Model

When you create a protected model, you can customize its behavior by defining callbacks. Callbacks specify code that executes when a protected model user views, simulates, or generates code for the protected model. A protected model user cannot view or modify a callback. To create a protected model with callbacks:

- 1 Define `Simulink.ProtectedModel.Callback` objects for each callback.
- 2 To create your protected model, call the `Simulink.ModelReference.protect` function. Use the `'Callbacks'` option to specify a cell array of callbacks to include in the protected model.

In this section...

“Creating Callbacks” on page 11-26

“Defining Callback Code” on page 11-27

“Create a Protected Model with Callbacks” on page 11-27

Creating Callbacks

To create and define a protected model callback, create a `Simulink.ProtectedModel.Callback` object. Callback objects specify:

- The code to execute for the callback. The code can be a character vector of MATLAB commands or a script on the MATLAB path.
- The event that triggers the callback. The event can be `'PreAccess'` or `'Build'`.
- The protected model functionality that the event applies to. The functionality can be `'CODEGEN'`, `'SIM'`, `'VIEW'`, or `'AUTO'`. If you select `'AUTO'`, and the event is `'PreAccess'`, the callback applies to each functionality. If you select `'AUTO'`, and the event is `'Build'`, the callback applies only to `'CODEGEN'` functionality. If you do not select any functionality, the default behavior is `'AUTO'`.
- The option to override the protected model build process. This option applies only to `'CODEGEN'` functionality.

You can create only one callback per event and per functionality.

Defining Callback Code

You can define the code for a callback by using either a character vector of MATLAB commands or a script on the MATLAB path. When you write callback code, follow these guidelines:

- Callbacks must use MATLAB code (.m or .p).
- The code can include protected model functions or any MATLAB command that does not require loading the model.
- Callback code must not call out to external utilities unless those utilities are available in the protected model user's environment.
- Callback code cannot reference the source protected model unless you are using protected model functions.

You can use the `Simulink.ProtectedModel.getCallbackInfo` function in callback code to get information on the protected model. The function returns a `Simulink.ProtectedModel.CallbackInfo` object that provides the protected model name and the names of submodels. If the callback is specified for 'CODEGEN' functionality and 'Build' event, the object provides the target identifier and model code interface type ('Top model' or 'Model reference').

Create a Protected Model with Callbacks

This example creates a protected model with a callback for code generation.

- 1 On the MATLAB path, create a callback script, `pm_callback.m`, containing:

```
s1 = 'Code interface is: ';
cbinfoobj = Simulink.ProtectedModel.getCallbackInfo(...
    'sldemo_mdhref_counter', 'Build', 'CODEGEN');
disp([s1 cbinfoobj.CodeInterface]);
```

- 2 Create a callback that uses the script. If the callback code replaces the protected model build process, set the override option.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
    'CODEGEN', 'pm_callback.m');
pmCallback.setOverrideBuild(true);
```

- 3 Create the protected model and specify the code generation callback.

```
Simulink.ModelReference.protect('sldemo_mdhref_counter',...
```

```
'Mode', 'CodeGeneration', 'Callbacks', {pmCallback})
```

- 4 Build the protected model. In place of the build, the callback displays the code interface.

```
rtwbuild('sldemo_mdref_basic')
```

See Also

[Simulink.ProtectedModel.Callback](#) | [Simulink.ModelReference.protect](#) | [Simulink.ProtectedModel.getCallbackInfo](#)

More About

- “Protect a Referenced Model” on page 11-2
- “Code Generation Support in a Protected Model” on page 11-6

Component Initialization, Reset, and Termination in Simulink Coder

Generate Code That Responds to Initialize, Reset, and Terminate Events

To generate code from a modeling component that responds to initialize, reset, and terminate events during execution, use the blocks **Initialize Function** and **Terminate Function**. For information on how to use these blocks, see “Create a Model to Initialize and Terminate State”. You can use the blocks anywhere in a model hierarchy.

Examples of when to generate code that responds to initialize, reset, or terminate events include:

- Starting and stopping a component.
- Calculating initial conditions.
- Saving and restoring state from nonvolatile memory.
- Generating reset entry-point functions that respond to external events.

Each nonvirtual subsystem and referenced model can have its own set of initialize, reset, and terminate functions.

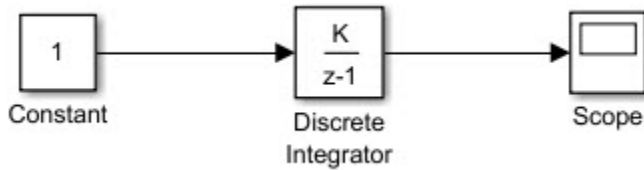
The code generator produces initialization and termination code differently than reset code. For initialization and termination code, the code generator includes your component initialization and termination code in the default entry-point functions, *model_initialize* and *model_terminate*. The code generator produces reset code only if you model reset behavior.

Generate Code for Initialize and Terminate Events

When you generate code for a component that includes **Initialize Function** and **Terminate Function** blocks, the code generator:

- Includes initialize event code with default initialize code in entry-point function *model_initialize*.
- Includes terminate event code with default terminate code in entry-point function *model_terminate*.

Consider the model `matlab/help/toolbox/simulink/examples/ex_model01.slx`.



For this model, the code generator produces initialize and terminate entry-point functions that other code can interface with.

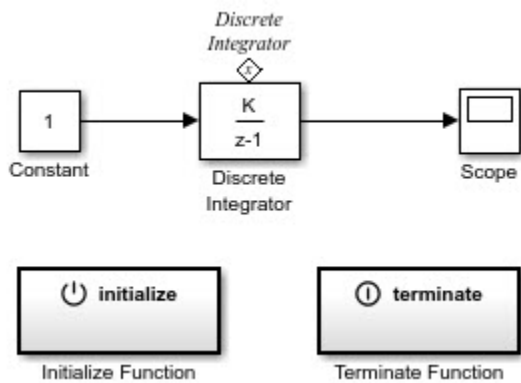
```
void ex_model01_initialize(void)
void ex_model01_terminate(void)
```

This code appears in the generated file `ex_model01.c`. The initialize function, `ex_model01_initialize`, initializes an error status. The terminate function, `ex_model01_terminate`, requires no code.

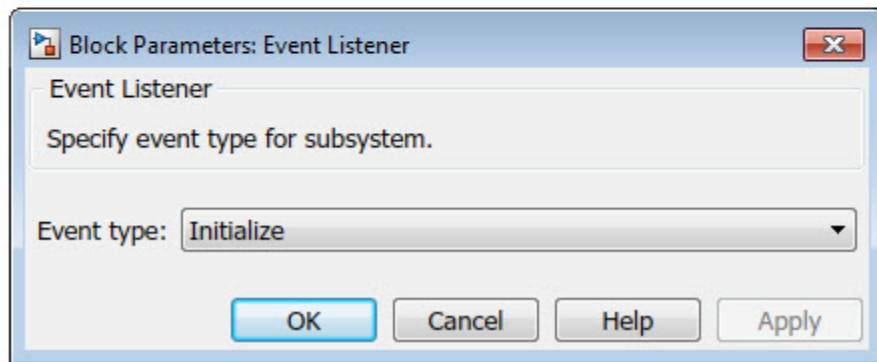
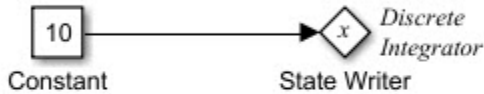
```
void ex_model01_initialize(void)
{
    rtmSetErrorStatus(ex_model01_M, (NULL));
}

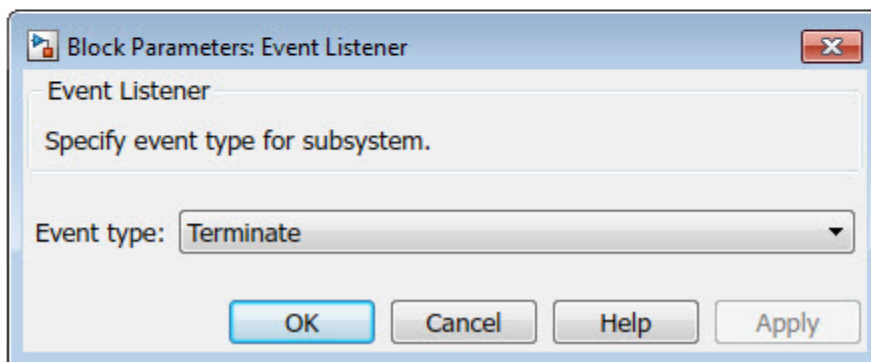
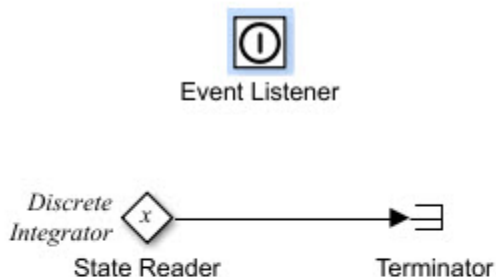
void ex_model01_terminate(void)
{
    /* (no terminate code required) */
}
```

Add **Initialize Function** and **Terminate Function** blocks to the model (see `matlab/help/toolbox/simulink/examples/ex_model01_initterm.slx`). The **Initialize Function** block uses the **State Writer** block to set the initial condition of a **Discrete Integrator** block. The **Terminate Function** block includes a **State Reader** block, which reads the state of the **Discrete Integrator** block.



The **Event type** parameter of the Event Listener block for the initialize and terminate functions is set to **Initialize** and **Terminate**, respectively. The initialize function uses the State Writer block to initialize the input value for the Discrete Integrator block to 10. The terminate function uses the State Reader block to read the state of the Discrete Integrator block.





The code generator includes the event code that it produces for the Initialize Function and Terminate Function blocks with standard initialize and terminate code in entry-point functions `ex_model01_initterm_initialize` and `ex_model01_initterm_terminate`. This code assumes that support for nonfinite numbers and MAT-file logging is disabled.

```
void ex_model01_initterm_initialize(void)
{
    rtmSetErrorStatus(ex_model01_initterm_M, (NULL));

    (void) memset(((void *) &ex_model01_initterm_B), 0,
        sizeof(B_ex_model01_initterm_T));

    (void) memset((void *)&ex_model01_initterm_DW, 0,
        sizeof(DW_ex_model01_initterm_T));

    ex_model01_initterm_DW.DiscreteIntegrator_DSTATE =
        ex_model01_initterm_P.Constant1_Value;
}
```

```
void ex_model01_initterm_terminate(void)
{
    ex_model01_initterm_B.read_state =
    ex_model01_initterm_DW.DiscreteIntegrator_DSTATE;
}
```

The initialization code:

- Initializes an error status.
- Allocates memory for block I/O and state parameters.
- Assigns the value of the constant input parameter to the state parameter of the discrete integrator.

The termination code assigns the value of the discrete integrator state parameter to the block I/O parameter.

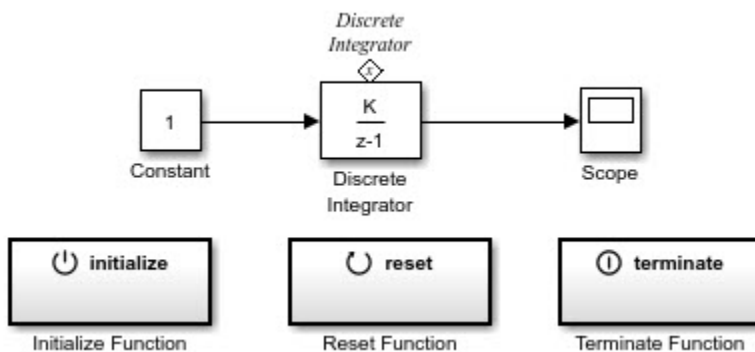
Generate Code for Reset Events

Generate code that responds to a reset event by including an **Initialize Function** or **Terminate Function** block in a modeling component. Configure the block for a reset by setting the **Event type** parameter of its **Event Listener** block to **Reset**. Also set the **Event name** parameter. The default name is **reset**.

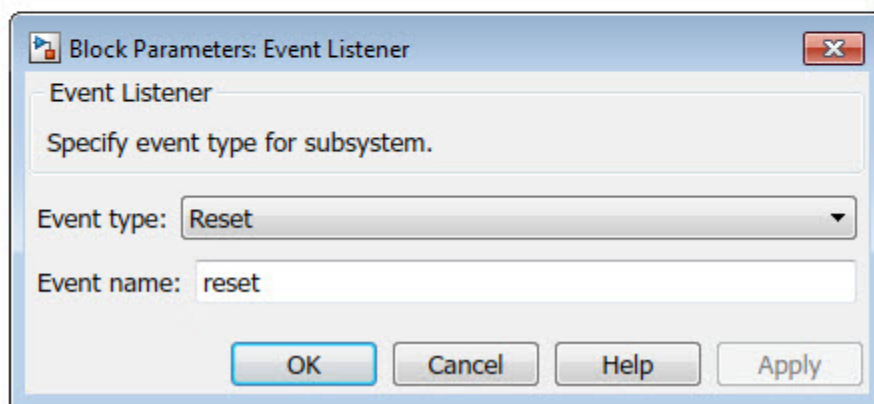
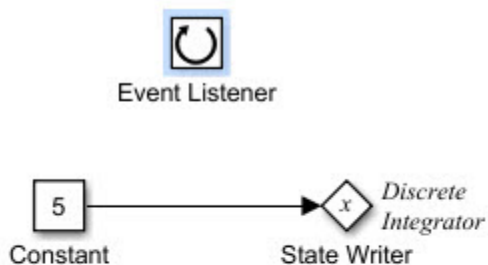
The code generator produces a reset entry-point function *only* if you model reset behavior. If a component contains multiple reset specifications, the code that the code generator produces depends on whether reset functions share an event name. For a given component hierarchy:

- For reset functions with unique event names, the code generator produces a separate entry-point function for each named event. The name of each function is the name of the corresponding event.
- For reset functions that share an event name, the code generator aggregates the reset code into one entry-point function. The code for the reset functions appears in order, starting with the lowest level (innermost) of the component hierarchy and ending with the root (outermost). The name of the function is *model_reset*. For more information, see “Event Names and Code Aggregation” on page 12-9.

Consider the model `matlab/help/toolbox/simulink/examples/ex_model01_irt.slx`, which includes a **Reset Function** block derived from an **Initialize Function** block.



The **Event type** and **Event name** parameters of the Event Listener block are set to Reset and reset, respectively. The function uses the State Writer block to reset the input value for the Discrete Integrator block to 5.



The code generator produces reset function `ex_model01_irt_reset`. The code for the function appears in the `Real-time model` section of the `model.c` file.

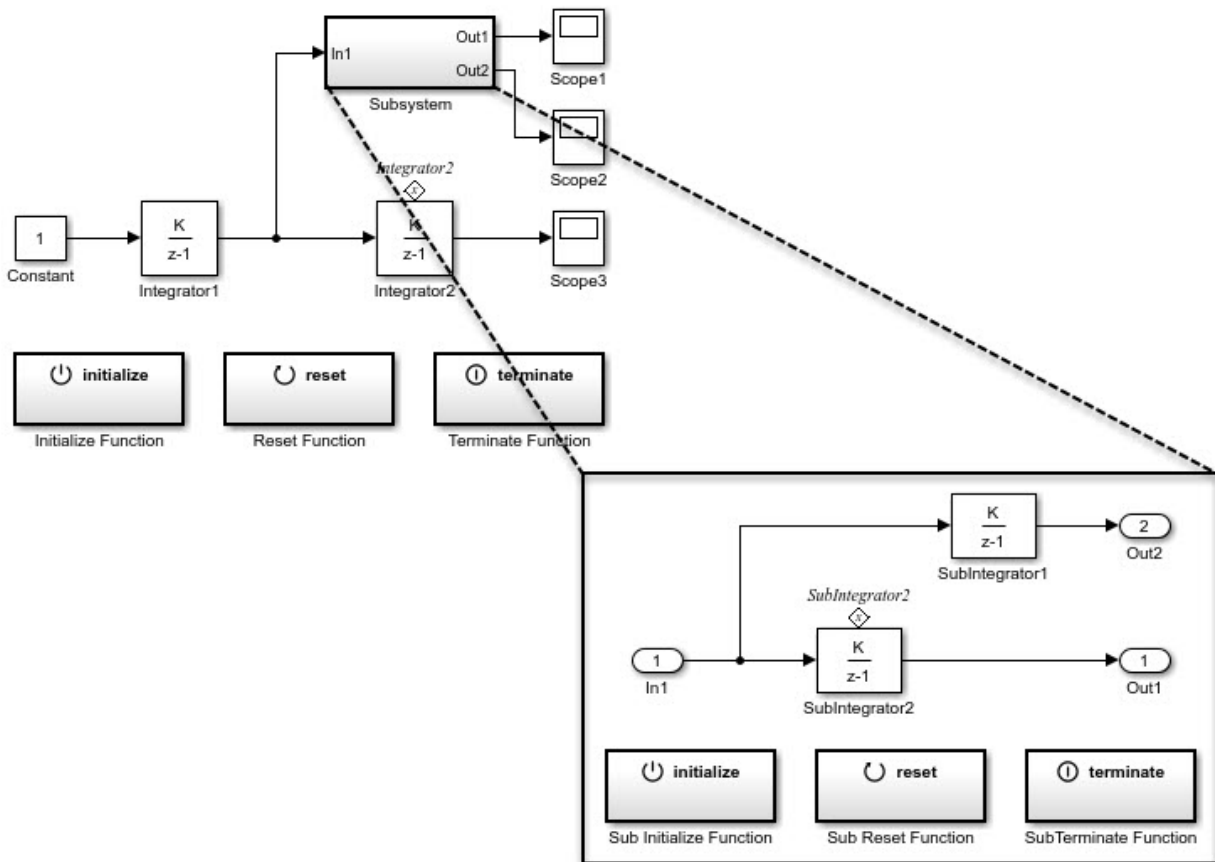
```
void ex_model01_irt_reset(void)
{
    ex_model01_irt_DW.DiscreteInterator_DSTATE =
        ex_model01_irt_P.Constant1_Value_b;
}
```

Event Names and Code Aggregation

Use the `Initialize Function` and `Terminate Function` blocks to define multiple initialize, reset, and terminate functions for a component hierarchy. Define only one initialize function and one terminate function per hierarchy level. You can define multiple reset functions for a hierarchy level. The event names that you configure for the functions at a given level must be unique.

When producing code, the code generator aggregates code for functions that have a given event name across the entire component hierarchy into one entry-point function. The code for reset functions appears in order, starting with the lowest level (innermost) of the component hierarchy and ending with the root (outermost). The code generator uses the event name to name the function.

For example, the model `matlab/help/toolbox/simulink/examples/ex_model01_shared_irt.slx` includes a subsystem that replicates the initialize, reset, and terminate functions that are in the parent model.



Although the model includes multiple copies of the initialize, reset, and terminate functions, the code generator produces one entry-point function for reset (`ex_model01_shared_irt_reset`), one for initialize (`ex_model01_shared_irt_initialize`), and one for terminate (`ex_model01_shared_irt_terminate`). Within each entry-point function, after listing code for blocks configured with an initial condition (`model_P.block_IC`), the code generator orders code for components, starting with the lowest level of the hierarchy and ending with the root.

⋮

```

.
void ex_model01_shared_irt_reset(void)
{
    ex_model01_shared_irt_DW.SubIntegrator2_DSTATE =
        ex_model01_shared_irt_P.Constant1_Value;

    ex_model01_shared_irt_DW.Integrator2_DSTATE =
        ex_model01_shared_irt_P.Constant_Value_b;
}
.
.
.
void ex_model01_shared_irt_initialize(void)
{
    rtmSetErrorStatus(ex_model01_shared_irt_M, (NULL));

    (void) memset(((void *) &ex_model01_shared_irt_DW), 0,
        sizeof(DW_ex_model01_shared_irt_T));

    ex_model01_shared_irt_DW.Integrator1_DSTATE =
        ex_model01_shared_irt_P.Integrator1_IC;

    ex_model01_shared_irt_DW.SubIntegrator2_DSTATE =
        ex_model01_shared_irt_P.Constant_Value_j;

    ex_model01_shared_irt_DW.Integrator2_DSTATE =
        ex_model01_shared_irt_P.Constant_Value;
.
.
.
void ex_model01_shared_irt_terminate(void)
{
    /* (no terminate code required) */

}

```

If you rename the event configured for the subsystem reset function to `reset_02`, the code generator produces two reset entry-point functions, `ex_model01_shared_irt_reset` and `ex_model01_shared_irt_reset_02`.

```

void ex_model01_share_irt_reset(void)
{
    ex_model01_shared_irt_DW.DiscreteIntegrator2_DSTATE =
        ex_model01_shared_irt_P.Constant1_Value_b;
}

```

```
}  
  
void ex_model01_share_irt_reset_02(void)  
{  
    ex_model01_shared_irt_DW.DiscreteIntegrator_DSTATE =  
        ex_model01_shared_irt_P.Constant1_Value;  
}
```

Limitations

You cannot generate code from a:

- Harness model—a root model that contains a Model block, which exposes initialize, reset, or terminate function ports.
- Model configured for C++ code generation.

Related Examples

- “Create a Model to Initialize and Terminate State”
- “Entry-Point Functions and Scheduling” on page 15-2

Stateflow Blocks in Simulink Coder

- “Code Generation of Stateflow Blocks” on page 13-2
- “Generate Reusable Code for Atomic Subcharts” on page 13-6
- “Generate Reusable Code for Unit Testing” on page 13-8
- “Inline State Functions in Generated Code” on page 13-14
- “Air-Fuel Ratio Control System with Stateflow Charts” on page 13-17

Code Generation of Stateflow Blocks

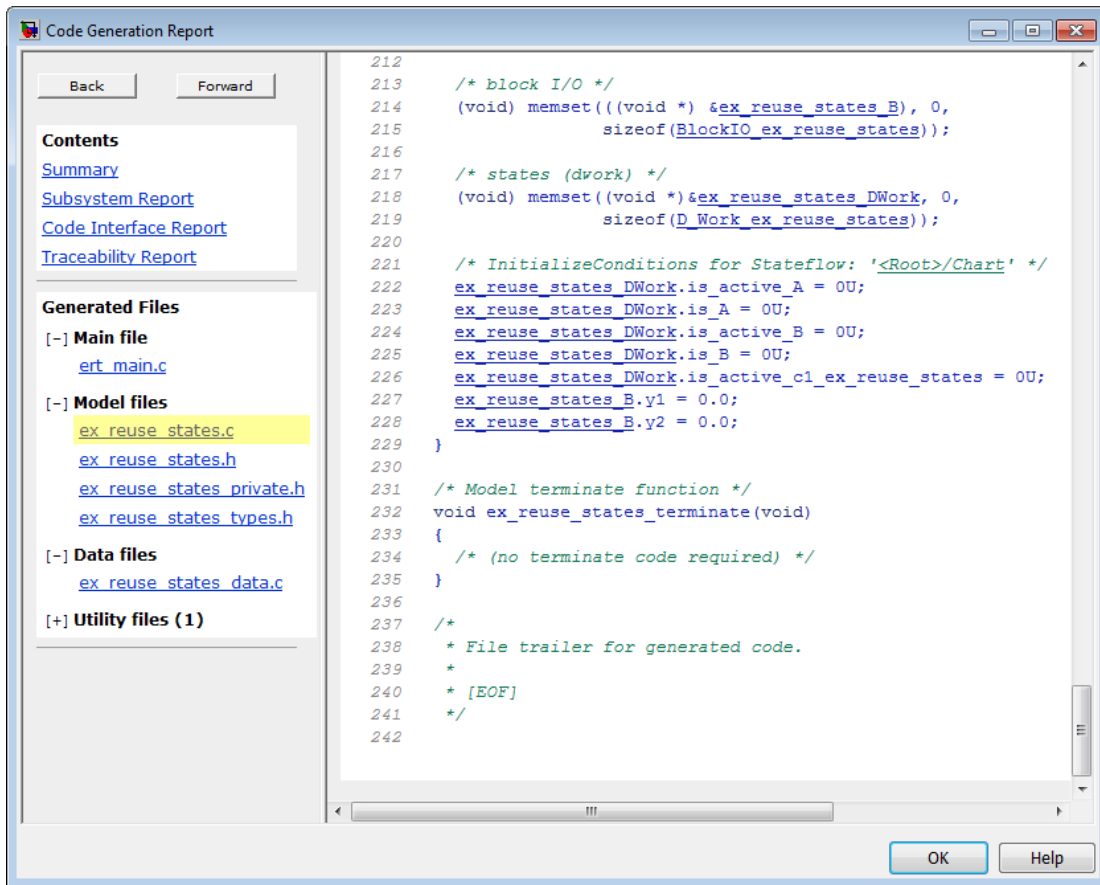
The code generator produces code for Stateflow blocks for rapid prototyping. If you have an Embedded Coder license, you can generate production code for Stateflow blocks.

Comparison of Code Generation Methods

The following sections compare two ways of generating code.

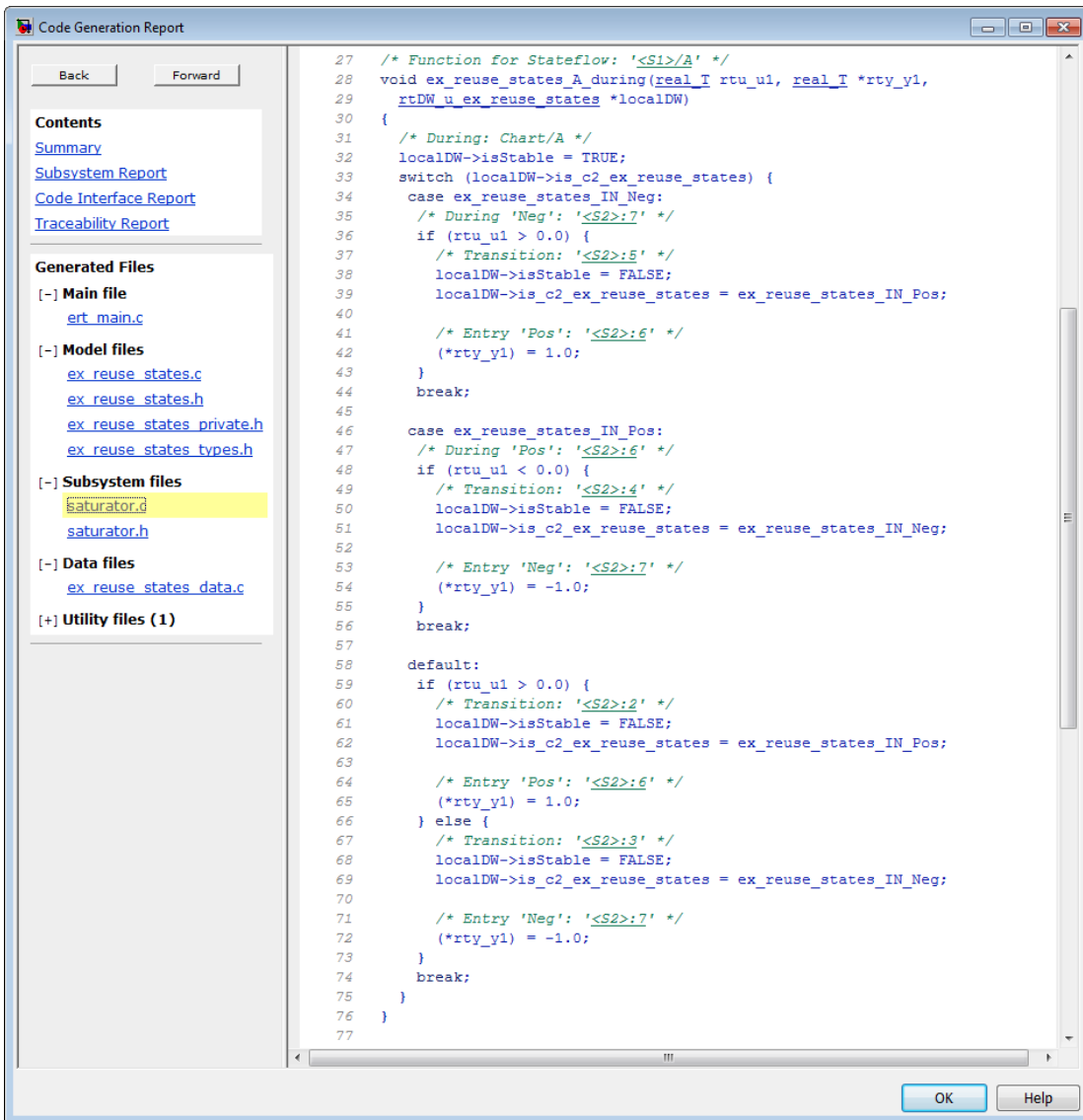
Code Generation Without Atomic Subcharts

You generate code for the entire model in one file and look through that entire file to find code for a specific part of the chart.



Code Generation With Atomic Subcharts

You specify code generation parameters so that code for an atomic subchart appears in a separate file. This method of code generation enables unit testing for a specific part of a chart. You can avoid searching through unrelated code and focus only on the part that interests you.



Note: Unreachable Stateflow states are optimized out and are not included in the generated code.

For more information, see “Generate Reusable Code for Unit Testing” on page 13-8.

Generate Reusable Code for Atomic Subcharts

In this section...

“How to Generate Reusable Code for Linked Atomic Subcharts” on page 13-6

“How to Generate Reusable Code for Unlinked Atomic Subcharts” on page 13-7

How to Generate Reusable Code for Linked Atomic Subcharts

To specify code generation parameters for linked atomic subcharts from the same library:

- 1 Open the library model that contains your atomic subchart.
- 2 Unlock the library.
- 3 Right-click the library chart and select **Block Parameters**.
- 4 In the dialog box, specify the following parameters:
 - a On the **Main** tab, select **Treat as atomic unit**.
 - b On the **Code Generation** tab, set **Function packaging** to **Reusable function**.
 - c Set **File name options** to **User specified**.
 - d For **File name**, enter the name of the file with no extension.
 - e Click **OK** to apply the changes.
- 5 (OPTIONAL) Customize the generated function names for atomic subcharts:
 - a Open the Model Configuration Parameters dialog box.
 - b On the **Code Generation** pane, set **System target file** to `ert.tlc`.
 - c Navigate to the **Code Generation > Symbols** pane.
 - d For **Subsystem methods**, specify the format of the function names using a combination of the following tokens:
 - `$R` — root model name
 - `$F` — type of interface function for the atomic subchart
 - `$N` — block name
 - `$H` — subsystem index
 - `$M` — name-mangling text

- e Click **OK** to apply the changes.

When you generate code for your model, a separate file stores the code for linked atomic subcharts from the same library.

How to Generate Reusable Code for Unlinked Atomic Subcharts

To specify code generation parameters for an unlinked atomic subchart:

- 1 In your chart, right-click the atomic subchart and select **Properties**.
- 2 In the dialog box, specify the following parameters:
 - a Set **Code generation function packaging** to `Reusable function`.
 - b Set **Code generation file name options** to `User specified`.
 - c For **Code generation file name**, enter the name of the file with no extension.
 - d Click **OK** to apply the changes.
- 3 (OPTIONAL) Customize the generated function names for atomic subcharts:
 - a Open the Model Configuration Parameters dialog box.
 - b On the **Code Generation** pane, set **System target file** to `ert.tlc`.
 - c Navigate to the **Code Generation > Symbols** pane.
 - d For **Subsystem methods**, specify the format of the function names using a combination of the following tokens:
 - `$R` — root model name
 - `$F` — type of interface function for the atomic subchart
 - `$N` — block name
 - `$H` — subsystem index
 - `$M` — name-mangling text
 - e Click **OK** to apply the changes.

When you generate code for your model, a separate file stores the code for the atomic subchart. For more information, see “Generate Reusable Code for Unit Testing” on page 13-8.

Generate Reusable Code for Unit Testing

In this section...

“Goal of the Tutorial” on page 13-8

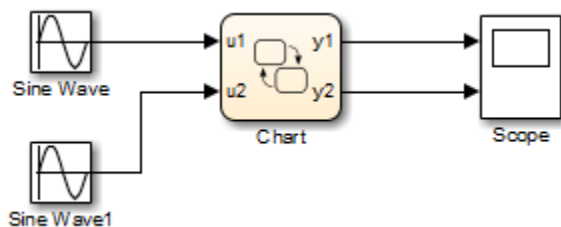
“Convert a State to an Atomic Subchart” on page 13-9

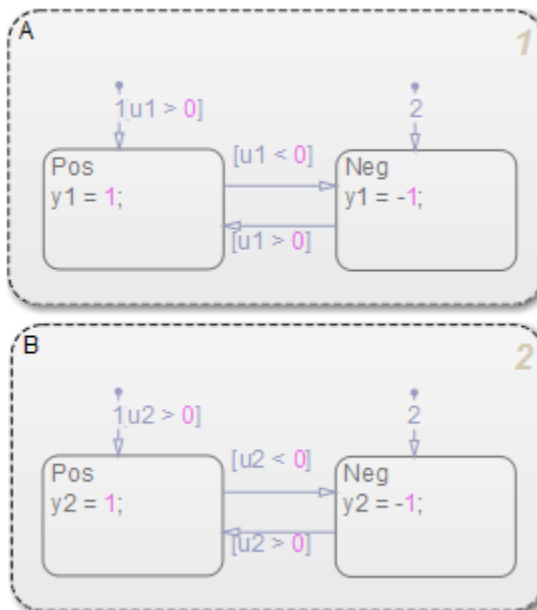
“Specify Code Generation Parameters” on page 13-10

“Generate Code for Only the Atomic Subchart” on page 13-11

Goal of the Tutorial

Assume that you have the following model, and the chart has two states:

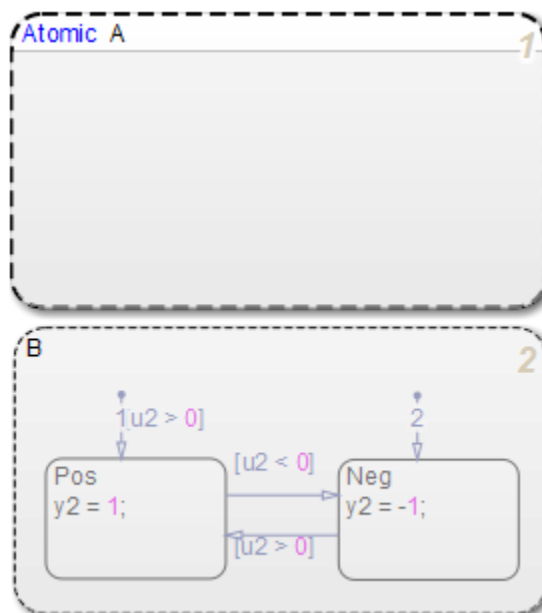




Suppose that you want to generate reusable code so that you can perform unit testing on state A. You can convert that part of the chart to an atomic subchart and then specify a separate file to store the generated code.

Convert a State to an Atomic Subchart

To convert state A to an atomic subchart, right-click the state and select **Group & Subchart > Atomic Subchart**. State A changes to an atomic subchart:



Specify Code Generation Parameters

Set Up a Standalone C File for the Atomic Subchart

- 1 Open the properties dialog box for A.
- 2 Set **Code generation function packaging** to Reusable function.
- 3 Set **Code generation file name options** to User specified.
- 4 For **Code generation file name**, enter saturator as the name of the file.
- 5 Click **OK**.

Set Up the Code Generation Report

- 1 Open the Model Configuration Parameters dialog box.
- 2 In the **Code Generation** pane, set **System target file** to ert.tlc.
- 3 In the **Code Generation > Report** pane, select **Create code generation report**.

This step automatically selects **Open report automatically** and **Code-to-model** on the **All Parameters** tab.

- 4 Select **Model-to-code** on the **All Parameters** tab.
- 5 Click **Apply**.

Customize the Generated Function Names

- 1 In the Model Configuration Parameters dialog box, go to the **Code Generation > Symbols** pane.
- 2 Set **Subsystem methods** to the format scheme $\$R\$N\$M\F , where:
 - $\$R$ is the root model name.
 - $\$N$ is the block name.
 - $\$M$ is the mangle token.
 - $\$F$ is the type of interface function for the atomic subchart.

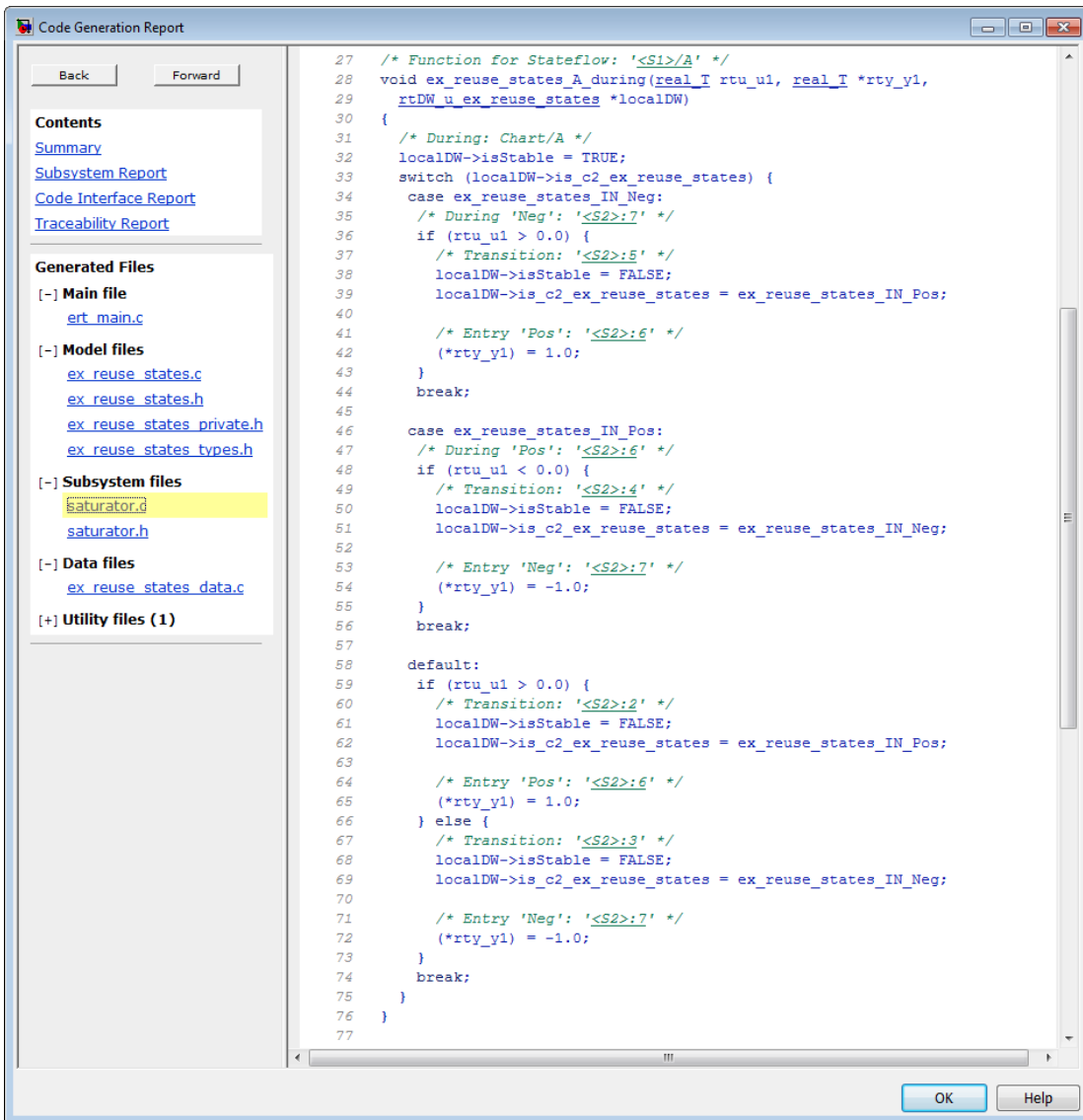
For more information, see “Subsystem methods”.

- 3 Click **Apply**.

Generate Code for Only the Atomic Subchart

To generate code for your model, press **Ctrl+B**. In the code generation report that appears, you see a separate file that contains the generated code for the atomic subchart.

To inspect the code for `saturator.c`, click the hyperlink in the report to see the following code:



Line 28 shows that the during function generated for the atomic subchart has the name `ex_reuse_states_A_during`. This name follows the format scheme `RNMF` specified for **Subsystem methods**:

- \$R is the root model name, `ex_reuse_states`.
- \$N is the block name, `A`.
- \$M is the mangle token, which is empty.
- \$F is the type of interface function for the atomic subchart, `during`.

Note: The line numbers shown can differ from the numbers that appear in your code generation report.

Inline State Functions in Generated Code

In this section...

“Inlined Generated Code for State Functions” on page 13-14

“How to Set the State Function Inline Option” on page 13-16

“Best Practices for Controlling State Function Inlining” on page 13-16

Inlined Generated Code for State Functions

By default, the code generator uses an internal heuristic to determine whether to inline generated code for state functions. The heuristic takes into consideration an inlining threshold. As code grows and shrinks in size, generated code for state functions can be unpredictable.

If your model includes Stateflow objects and you have rigorous requirements for traceability between generated code and the corresponding state functions, you can override the default behavior. Use the state property `Function Inline Option` to explicitly force or prevent inlining of state functions.

What Happens When You Force Inlining

If you force inlining for a state, the code generator inlines code for state actions into the parent function. The parent function contains code for executing the state actions, outer transitions, and flow charts. It does not include code for empty state actions.

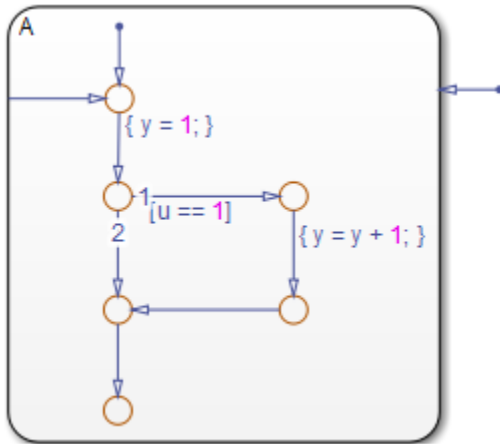
What Happens When You Prevent Inlining

If you prevent inlining for a state, the code generator produces these static functions for state *foo*.

Function	Description
<code>enter_atomic_foo</code>	Marks <i>foo</i> active and performs entry actions.
<code>enter_internal_foo</code>	Calls default paths.
<code>inner_default_foo</code>	Executes flow charts that originate when an inner transition and default transition reach the same junction inside a state.

Function	Description
	<p>The code generator produces this function only when the flow chart is complex enough to exceed the inlining threshold.</p> <p>In generated code, Stateflow software calls this function from both the <code>enter_internal_foo</code> and <code>foo</code> functions.</p>
<code>foo</code>	Checks for valid outer transitions and if none, performs during actions.
<code>exit_atomic_foo</code>	Performs exit actions and marks <code>foo</code> inactive.
<code>exit_internal_foo</code>	Performs exit actions of the child substates and then exits <code>foo</code> .

Suppose the following chart is in model M.



If you prevent inlining for state A, the code generator produces this code.

```

static void M_inner_default_A(void);
static void M_exit_atomic_A(void);
static void M_A(void);
static void M_enter_atomic_A(void);
static void M_enter_internal_A(void);
  
```

How to Set the State Function Inline Option

To set the function inlining property for a state:

- 1 Right-click inside the state and, from the context menu, select **Properties**.

The State properties dialog box opens.

- 2 In the **Function Inline Option** field, select one of these options.

Option	Behavior
Inline	Forces inlining of state functions into the parent function, as long as the function is not part of a recursion. See “What Happens When You Force Inlining” on page 13-14.
Function	Prevents inlining of state functions. Generates up to six static functions for the state. See “What Happens When You Prevent Inlining” on page 13-14.
Auto	Uses internal heuristics to determine whether or not to inline the state functions.

- 3 Click **Apply**.

Best Practices for Controlling State Function Inlining

To	Set Function Inline Option Property To
Generate a separate function for each action of a state and a separate function for each action of its substates	Function for the state and each substate
Generate a separate function for each action of a state, but include code for the associated action of its substates	Function for the state and Inline for each substate

Air-Fuel Ratio Control System with Stateflow Charts

Generate code for an air-fuel ratio control system designed with Simulink® and Stateflow®.

Figures 1, 2, and 3 show relevant portions of the `sldemo_fuelsys` model, a closed-loop system containing a plant and controller. The plant validates the controller in simulation early in the design cycle. In this example, you generate code for the relevant controller subsystem, "fuel_rate_control". Figure 1 shows the top-level simulation model.

Open `sldemo_fuelsys` via `rtwdemo_fuelsys` and compile the diagram to see the signal data types.

```
rtwdemo_fuelsys
sldemo_fuelsys([],[],[],'compile');
sldemo_fuelsys([],[],[],'term');
```

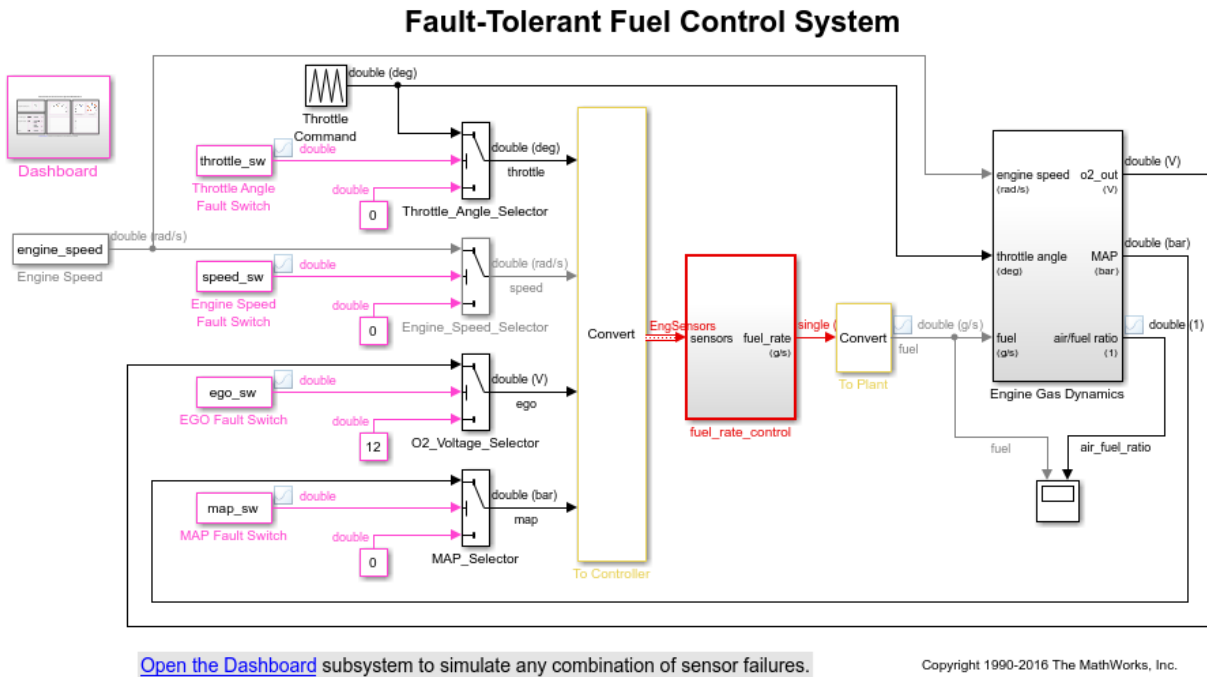


Figure 1: Top-level model of the plant and controller

The air-fuel ratio control system is comprised of Simulink® and Stateflow®. The control system is the portion of the model for which you generate code.

```
open_system('sldemo_fuelsys/fuel_rate_control');
```

Fuel Rate Control Subsystem

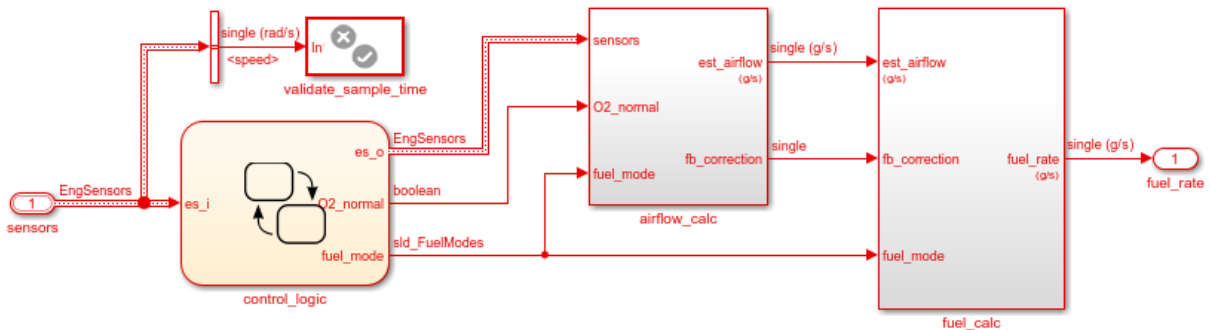


Figure 2: The air-fuel ratio controller subsystem

The control logic is a Stateflow® chart that specifies the different modes of operation.

```
open_system('sldemo_fuelsys/fuel_rate_control/control_logic');
```

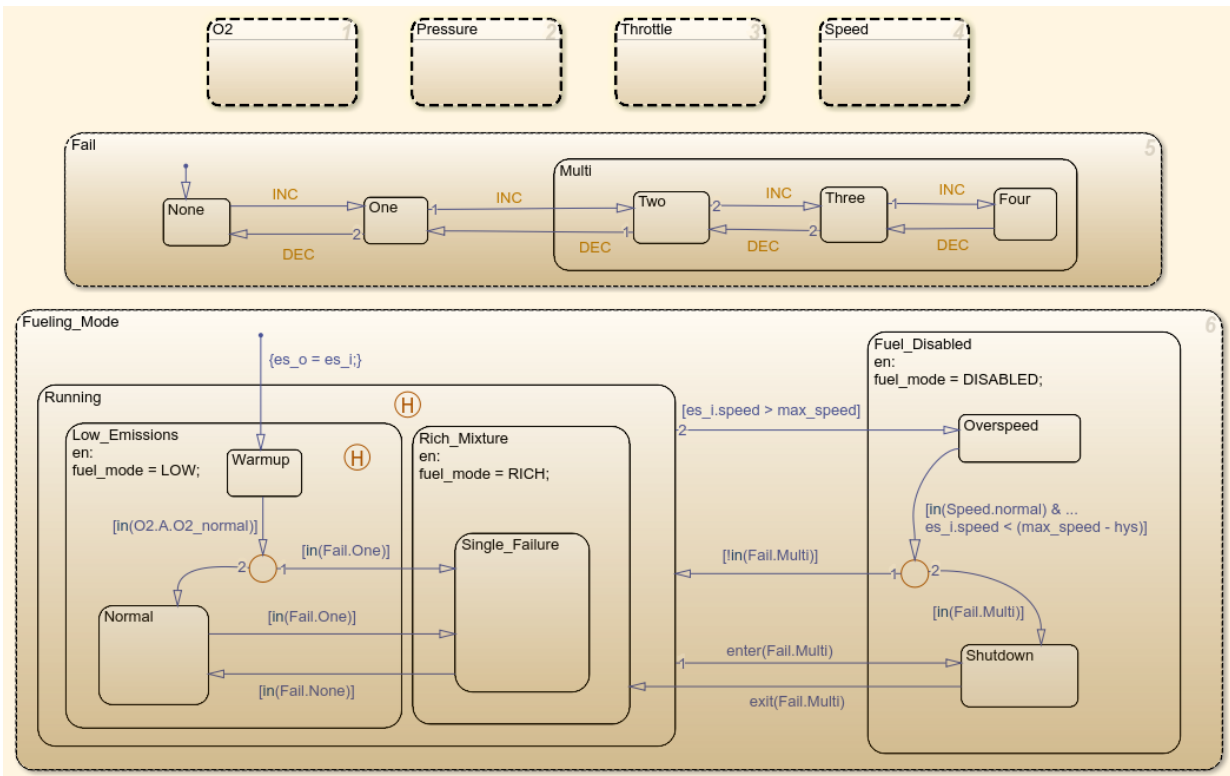


Figure 3: Air-fuel rate controller logic

Close these windows.

```
close_system('sldemo_fuelsys/fuel_rate_control/airflow_calc');
close_system('sldemo_fuelsys/fuel_rate_control/fuel_calc');
close_system('sldemo_fuelsys/fuel_rate_control/control_logic');
hDemo.rt=sfroot;hDemo.m=hDemo.rt.find('-isa','Simulink.BlockDiagram');
hDemo.c=hDemo.m.find('-isa','Stateflow.Chart','-and','Name','control_logic');
hDemo.c.visible=false;
close_system('sldemo_fuelsys/fuel_rate_control');
```

Configure and Build the Model with Simulink® Coder™

Simulink® Coder™ generates generic ANSI® C code for Simulink® and Stateflow® models via the Generic Real-Time (GRT) target. You can configure a model for code generation programmatically.

```
rtwconfiguredemo('sldemo_fuelsys','GRT');
```

For this example, build only the air-fuel ratio control system. Once the code generation process is complete, an HTML report detailing the generated code is displayed. The main body of the code is located in `fuel_rate_control.c`.

```
rtwbuild('sldemo_fuelsys/fuel_rate_control');

### Starting build procedure for model: fuel_rate_control
### Successful completion of build procedure for model: fuel_rate_control
```

Configure and Build the Model with Embedded Coder®

Embedded Coder® generates production ANSI® C/C++ code via the Embedded Real-Time (ERT) target. You can configure a model for code generation programmatically.

```
rtwconfiguredemo('sldemo_fuelsys','ERT');
```

Repeat the build process and inspect the generated code. In the Simulink® Coder™ Report, you can navigate to the relevant code segments interactively by using the **Previous** and **Next** buttons. From the chart context menu (right-click the Stateflow® block), select **Code Generation > Navigate to Code**. Programmatically, use the `rtwtrace` utility.

```
rtwbuild('sldemo_fuelsys/fuel_rate_control');
rtwtrace('sldemo_fuelsys/fuel_rate_control/control_logic')

### Starting build procedure for model: fuel_rate_control
### Successful completion of build procedure for model: fuel_rate_control
```

View the air-fuel ratio control logic in the generated code.

```
rtwdemodbtype('fuel_rate_control_ert_rtw/fuel_rate_control.c','/* Function for Chart: '

/* Function for Chart: '<S1>/control_logic' */
static void Fueling_Mode(const int32_T *sfEvent)
{
    /* During 'Fueling_Mode': '<S3>:21' */
```

```

/* This state interprets the other states in the chart to directly control the fueling
switch (rtDW.bitsForTIDO.is_Fueling_Mode) {
case IN_Fuel_Disabled:
/* During 'Fuel_Disabled': '<S3>:22' */
/* The fuel is completely shut off while in this state. */
switch (rtDW.bitsForTIDO.is_Fuel_Disabled) {
case IN_Overspeed:
/* Inport: '<Root>/sensors' */
/* During 'Overspeed': '<S3>:24' */
/* The speed is dangerously high, so shut off the fuel. */
if ((rtDW.bitsForTIDO.is_Speed == IN_normal) && (rtU.sensors.speed <
603.0F)) {
/* Transition: '<S3>:54' */
if (!rtDW.bitsForTIDO.is_Fail == IN_Multi)) {
/* Transition: '<S3>:55' */
rtDW.bitsForTIDO.is_Fuel_Disabled = IN_NO_ACTIVE_CHILD;
rtDW.bitsForTIDO.is_Fueling_Mode = IN_Running;

/* Entry Internal 'Running': '<S3>:23' */
switch (rtDW.bitsForTIDO.was_Running) {
case IN_Low_Emissions:
if (rtDW.bitsForTIDO.is_Running != IN_Low_Emissions) {
rtDW.bitsForTIDO.is_Running = IN_Low_Emissions;
rtDW.bitsForTIDO.was_Running = IN_Low_Emissions;

/* Entry 'Low_Emissions': '<S3>:25' */
rtDW.fuel_mode = LOW;
}

/* Entry Internal 'Low_Emissions': '<S3>:25' */
switch (rtDW.bitsForTIDO.was_Low_Emissions) {
case IN_Normal:
rtDW.bitsForTIDO.is_Low_Emissions = IN_Normal;
rtDW.bitsForTIDO.was_Low_Emissions = IN_Normal;
break;
}
}
}

```

Close the model and code generation report.

```

clear hDemo;
rtwdemoclean;
close_system('sldemo_fuelsys',0);

```

Related Examples

For related fixed-point examples that use sldemo_fuelsys, see

- **Fixed-point design** - “Fixed-Point Fuel Rate Control System”
- **Fixed-point production C/C++ code generation** - “Air-Fuel Ratio Control System with Fixed-Point Data”

Data, Function, and File Definition

Data Representation in Simulink Coder

- “Access Signal, State, and Parameter Data During Execution” on page 14-3
- “Default Data Structures in the Generated Code” on page 14-16
- “Use the Real-Time Model Data Structure” on page 14-19
- “Use Enumerated Data in Generated Code” on page 14-22
- “Data Stores in Generated Code” on page 14-31
- “Structures in Generated Code Using Data Stores” on page 14-38
- “Specify Single Precision” on page 14-42
- “Block Parameter Representation in the Generated Code” on page 14-43
- “Configure Block Parameter Tunability for Rapid Prototyping” on page 14-52
- “Tune Phase Parameter of Sine Wave Block During Code Execution” on page 14-54
- “Create Tunable Calibration Parameter in the Generated Code” on page 14-56
- “Specify Instance-Specific Parameter Values for Reusable Referenced Model” on page 14-61
- “Parameter Data Types in the Generated Code” on page 14-76
- “Generate Efficient Code by Specifying Data Types for Block Parameters” on page 14-81
- “Reuse Parameter Data in Different Data Type Contexts” on page 14-90
- “Organize Block Parameter Values into Structures in the Generated Code” on page 14-94
- “Switch Between Sets of Parameter Values During Simulation and Code Execution” on page 14-100
- “Signal Representation in Generated Code” on page 14-109
- “Control Signals and States in Code by Applying Storage Classes” on page 14-120

- “Design Data Interface by Configuring Inport and Outport Blocks” on page 14-131
- “Group Signals into Structures in the Generated Code Using Buses” on page 14-136
- “Generate Efficient Code for Bus Signals” on page 14-140
- “Maximize Signal Storage Optimization” on page 14-144
- “Control Signal and State Initialization in the Generated Code” on page 14-145
- “Continuous Block State Naming in Generated Code” on page 14-156
- “Discrete Block State Naming in Generated Code” on page 14-158
- “Signal Processing with Fixed-Point Data” on page 14-163
- “Optimize Generated Code Using Fixed-Point Data with Simulink®, Stateflow®, and MATLAB®” on page 14-165
- “Declare Workspace Variables as Tunable Parameters Using the Model Parameter Configuration Dialog Box” on page 14-166

Access Signal, State, and Parameter Data During Execution

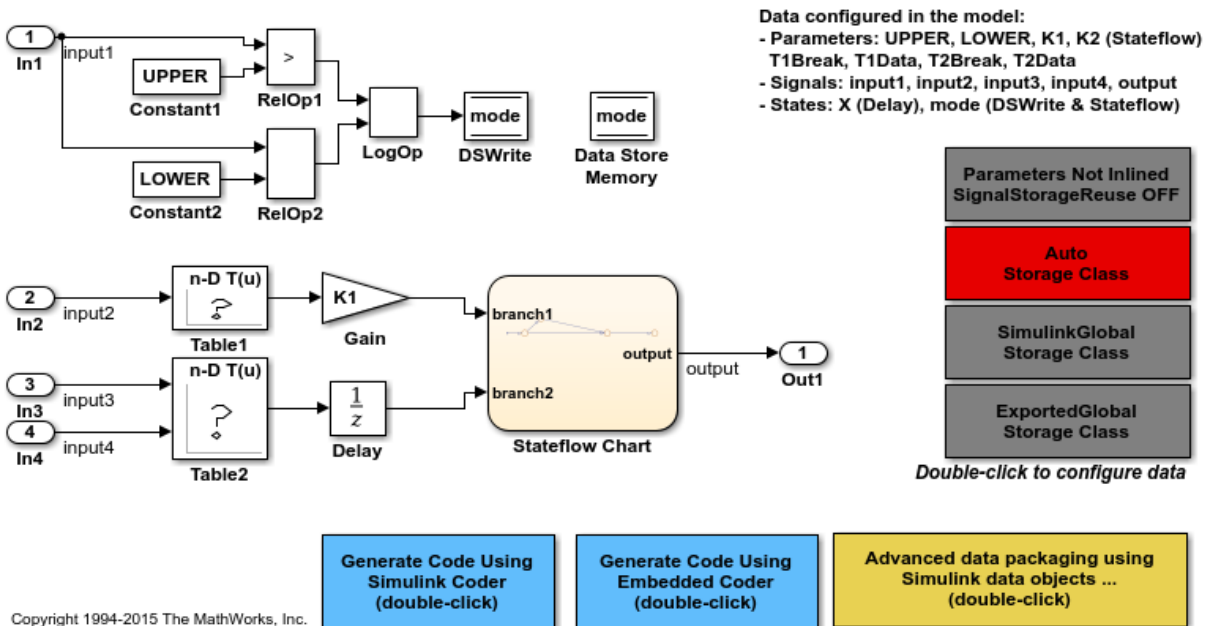
As you iteratively develop a model, you capture output signal and state data that model execution generates. You also tune parameter values during execution to observe the effect on the outputs. You can then base your design decisions upon analysis of these outputs. To access this signal, state, and parameter data in a rapid prototyping environment, you can configure the generated code to store the data in addressable memory.

By default, optimization settings make the generated code more efficient by eliminating unnecessary signal storage and inlining the numeric values of block parameters. To generate code that instead allocates addressable memory for this data, you can disable the optimizations or specify code generation settings for individual data items.

Explore Example Model

Open the example model `rtwdemo_basicsc`.

`rtwdemo_basicsc`



The model loads numeric MATLAB variables, such as K1, into the base workspace.

In the model, open the block dialog box for the Gain block labeled **Gain**. The block uses the variable K1 to set the value of the **Gain** parameter.

Disable Optimizations

In the model, clear the model configuration parameter **Signal storage reuse**. When you clear this optimization and other optimizations such as **Eliminate superfluous local variables (expression folding)**, the generated code allocates memory for signal lines. Clearing **Signal storage reuse** disables most of the other optimizations.

```
set_param('rtwdemo_basicsc','OptimizeBlockIOStorage','off')
```

Set the optimization **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** to Tunable. When set to Tunable, this configuration parameter causes the generated code to allocate memory for block parameters and workspace variables.

```
set_param('rtwdemo_basicsc','DefaultParameterBehavior','Tunable')
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')
```

```
### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

In the code generation report, view the file `rtwdemo_basicsc.h`. This header file defines a structure type that contains signal data. The structure contains fields that each represent a signal line in the model. For example, the output signal of the Gain block labeled **Gain** appears as the field **Gain**.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.h');
rtwdemodbtype(file,'/* Block signals (auto storage) */',...
    'B_rtwdemo_basicsc_T;',1,1)
```

```
/* Block signals (auto storage) */
typedef struct {
    real32_T Table1;          /* '<Root>/Table1' */
    real32_T Gain;           /* '<Root>/Gain' */
    real32_T Delay;         /* '<Root>/Delay' */
}
```

```

    real32_T Table2;          /* '<Root>/Table2' */
    boolean_T RelOp1;        /* '<Root>/RelOp1' */
    boolean_T RelOp2;        /* '<Root>/RelOp2' */
    boolean_T LogOp;         /* '<Root>/LogOp' */
} B_rtwdemo_basicsc_T;

```

The file defines a structure type that contains block parameter data. The MATLAB variable K1 appears as a field of the structure. The other fields of the structure represent other block parameters and workspace variables from the model, including initial conditions for signals.

```

rtwdemodbtype(file, /* Parameters (auto storage) */ , ...
    /* Real-time Model Data Structure */ , 1, 0)

```

```

/* Parameters (auto storage) */
struct P_rtwdemo_basicsc_T_ {
    real_T K2;                /* Variable: K2
                             * Referenced by: '<Root>/Stateflow Chart'
                             */
    real32_T LOWER;          /* Variable: LOWER
                             * Referenced by: '<Root>/Constant2'
                             */
    real32_T T1Break[11];    /* Variable: T1Break
                             * Referenced by: '<Root>/Table1'
                             */
    real32_T T1Data[11];     /* Variable: T1Data
                             * Referenced by: '<Root>/Table1'
                             */
    real32_T T2Break[3];     /* Variable: T2Break
                             * Referenced by: '<Root>/Table2'
                             */
    real32_T T2Data[9];      /* Variable: T2Data
                             * Referenced by: '<Root>/Table2'
                             */
    real32_T UPPER;          /* Variable: UPPER
                             * Referenced by: '<Root>/Constant1'
                             */
    int8_T K1;               /* Variable: K1
                             * Referenced by: '<Root>/Gain'
                             */
    real32_T Delay_InitialCondition; /* Computed Parameter: Delay_InitialCondition
                             * Referenced by: '<Root>/Delay'
                             */
    uint32_T Table2_maxIndex[2]; /* Computed Parameter: Table2_maxIndex

```

```

        * Referenced by: '<Root>/Table2'
        */
    boolean_T DataStoreMemory_InitialValue; /* Computed Parameter: DataStoreMemory_InitialValue
        * Referenced by: '<Root>/Data Store Memory'
        */
};

```

View the file `rtwdemo_basicsc_data.c`. This source file allocates global memory for a parameter structure and initializes the field values based on the parameter values in the model.

View the source file `rtwdemo_basicsc.c`. The code allocates global memory for a structure variable that contains signal data.

```

file = fullfile('rtwdemo_basicsc_grt_rtw', 'rtwdemo_basicsc.c');
rtwdemodbtype(file, /* Block signals (auto storage) */', ...
    'B_rtwdemo_basicsc_T rtwdemo_basicsc_B;', 1, 1)

/* Block signals (auto storage) */
B_rtwdemo_basicsc_T rtwdemo_basicsc_B;

```

The code algorithm in the model `step` function calculates the signal values. It then assigns these values to the fields of the signal structure. To perform the calculations, the algorithm uses the parameter values from the fields of the parameter structure.

Exclude Data Items from Optimizations

When you want to select code generation optimizations such as **Signal storage reuse**, you can preserve individual data items from the optimizations. The generated code then allocates addressable memory for the items.

Select the optimizations that you previously cleared.

```

set_param('rtwdemo_basicsc', 'OptimizeBlockIOStorage', 'on')
set_param('rtwdemo_basicsc', 'LocalBlockOutputs', 'on')
set_param('rtwdemo_basicsc', 'DefaultParameterBehavior', 'Inlined')

```

Right-click the output of the Gain block labeled **Gain** and select **Properties**. In the Signal Properties dialog box, select **Test point**.

```

portHandle = get_param('rtwdemo_basicsc/Gain', 'PortHandles');
portHandle = portHandle.Outport;
set_param(portHandle, 'TestPoint', 'on')

```

Convert the MATLAB variable `K1` to a `Simulink.Parameter` object. With parameter objects, you can create addressable parameters to tune during execution of the generated code.

```
K1 = Simulink.Parameter(K1);
```

Apply a storage class other than `Auto` to the parameter object `K1`. For example, use the storage class `SimulinkGlobal` to represent the parameter object as a field of the global parameter structure.

```
K1.StorageClass = 'SimulinkGlobal';
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')
```

```
### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

In the code generation report, view the file `rtwdemo_basicsc.h`. The structure that contains signal data now defines only one field, `Gain`, which represents the test-pointed output of the `Gain` block.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.h');
rtwdemodbtype(file, '/* Block signals (auto storage) */', ...
    'B_rtwdemo_basicsc_T;', 1, 1)
```

```
/* Block signals (auto storage) */
typedef struct {
    real32_T Gain; /* '<Root>/Gain' */
} B_rtwdemo_basicsc_T;
```

The structure that contains block parameter data defines one field, `K1`, which represents the parameter object `K1`.

```
rtwdemodbtype(file, '/* Parameters (auto storage) */', ...
    '/* Real-time Model Data Structure */', 1, 0)
```

```
/* Parameters (auto storage) */
struct P_rtwdemo_basicsc_T_ {
    int8_T K1; /* Variable: K1
               * Referenced by: '<Root>/Gain'
               */
};
```

Access Data Through Generated Interfaces

You can configure the generated code to contain extra code and files so that you can access model data through standardized interfaces. For example, use the C API to log signal data and tune parameters during execution.

Copy this custom source code into a file named `myHandCode.c` in your current folder.

```
#include "myHandHdr.h"

#define paramIdx 0 /* Index of the target parameter,
determined by inspecting the array of structures generated by the C API. */
#define sigIdx 0 /* Index of the target signal,
determined by inspecting the array of structures generated by the C API. */

void tuneFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr)
{
    /* Take action with the parameter value only at
    the beginning of simulation and at the 5-second mark. */
    if (*tPtr == 0 || *tPtr == 5) {

        /* Local variables to store information extracted from
        the model mapping information (mmi). */
        void** dataAddrMap;
        const rtwCAPI_DataTypeMap *dataTypeMap;
        const rtwCAPI_ModelParameters *params;
        int_T addrIdx;
        uint16_T dTypeIdx;
        uint8_T slDataType;

        /* Use built-in C API macros to extract information. */
        dataAddrMap = rtwCAPI_GetDataAddressMap(mmi);
        dataTypeMap = rtwCAPI_GetDataTypeMap(mmi);
        params = rtwCAPI_GetModelParameters(mmi);
        addrIdx = rtwCAPI_GetModelParameterAddrIdx(params,paramIdx);
        dTypeIdx = rtwCAPI_GetModelParameterDataTypeIdx(params,paramIdx);
        slDataType = rtwCAPI_GetDataTypeSLId(dataTypeMap, dTypeIdx);

        /* Handle data types 'double' and 'int8'. */
        switch (slDataType) {

            case SS_DOUBLE: {
                real_T* dataAddress;
```



```

        dataAddress = dataAddrMap[addrIdx];
        /* At the 5-second mark, increment the parameter value by 1. */
        if (*tPtr == 5) {
            (*dataAddress)++;
        }
        printf("Parameter value is %f\n", *dataAddress);
        break;
    }

    case SS_INT8: {
        int8_T* dataAddress;
        dataAddress = dataAddrMap[addrIdx];
        if (*tPtr == 5) {
            (*dataAddress)++;
        }
        printf("Parameter value is %i\n", *dataAddress);
        break;
    }
}
}

void logFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr)
{
    /* Take action with the signal value only when
       the simulation time is an integer value. */
    if (*tPtr-(int_T)*tPtr == 0) {

        /* Local variables to store information extracted from
           the model mapping information (mmi). */
        void** dataAddrMap;
        const rtwCAPI_DataTypeMap *dataTypeMap;
        const rtwCAPI_Signals *sigs;
        int_T addrIdx;
        uint16_T dTypeIdx;
        uint8_T slDataType;

        /* Use built-in C API macros to extract information. */
        dataAddrMap = rtwCAPI_GetDataAddressMap(mmi);
        dataTypeMap = rtwCAPI_GetDataTypeMap(mmi);
        sigs = rtwCAPI_GetSignals(mmi);
        addrIdx = rtwCAPI_GetSignalAddrIdx(sigs, sigIdx);
        dTypeIdx = rtwCAPI_GetSignalDataTypeIdx(sigs, sigIdx);
        slDataType = rtwCAPI_GetDataTypeSLId(dataTypeMap, dTypeIdx);
    }
}

```

```

/* Handle data types 'double' and 'single'. */
switch (slDataType) {

    case SS_DOUBLE: {
        real_T* dataAddress;
        dataAddress = dataAddrMap[addrIdx];
        printf("Signal value is %f\n", *dataAddress);
        break;
    }

    case SS_SINGLE: {
        real32_T* dataAddress;
        dataAddress = dataAddrMap[addrIdx];
        printf("Signal value is %f\n", *dataAddress);
        break;
    }
}
}
}

```

Copy this custom header code into a file named `myHandHdr.h` in your current folder.

```

#include <stdio.h>
#include <string.h>
#include <math.h>
/* Include rtw_modelmap.h for definitions of C API macros. */
#include "rtw_modelmap.h"
#include "builtin_typeid_types.h"
#include "rtwtypes.h"
void tuneFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr);
void logFcn(rtwCAPI_ModelMappingInfo *mmi, time_T *tPtr);

```

These files use the C API to access signal and parameter data in the code that you generate from the example model.

In the model, set **Configuration Parameters > Code Generation > Custom Code > Insert custom C code in generated > Header file** to `#include "myHandHdr.h"`. In the same pane in the Configuration Parameters dialog box, set **Additional Build Information > Source files** to `myHandCode.c`.

```

set_param('rtwdemo_basicsc', 'CustomHeaderCode', '#include "myHandHdr.h"')
set_param('rtwdemo_basicsc', 'CustomSource', 'myHandCode.c')

```

Select **Configuration Parameters > All Parameters > MAT-file Logging**. The generated executable runs only until the simulation stop time (which you set in the model configuration parameters).

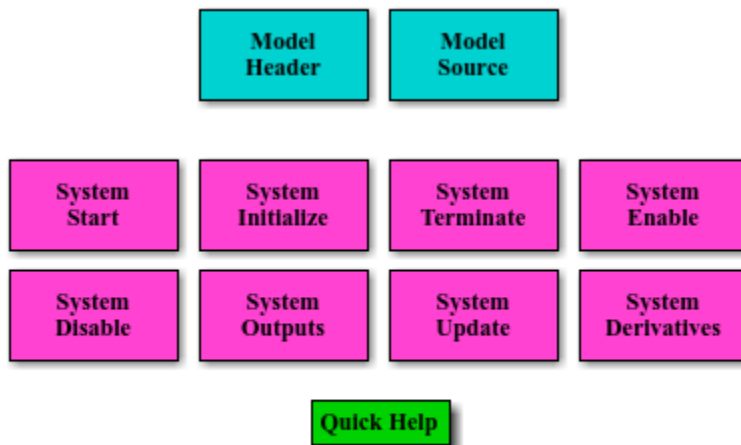
```
set_param('rtwdemo_basicsc','MatFileLogging','on')
```


Select all of the options under **Configuration Parameters > Code Generation > Interface > Generate C API for**.

```
set_param('rtwdemo_basicsc','RTWCAPIParams','on')
set_param('rtwdemo_basicsc','RTWCAPISignals','on')
set_param('rtwdemo_basicsc','RTWCAPIStates','on')
set_param('rtwdemo_basicsc','RTWCAPIRootIO','on')
```

Load the Custom Code block library.

```
custcode
```



 These blocks allow you to insert custom code into specific files and functions.

Add a System Outputs block to the model.

```
add_block('custcode/System Outputs','rtwdemo_basicsc/System Outputs')
```

In the System Outputs block dialog box, set **System Outputs Function Execution Code** to this custom code:

```

{
rtwdemo_basicsc_U.input2++;
rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_basicsc_M).mmi);
tuneFcn(MMI, rtmGetTPtr(rtwdemo_basicsc_M));
}

```

In the block dialog box, set **System Outputs Function Exit Code** to this custom code:

```

{
rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_basicsc_M).mmi);
logFcn(MMI, rtmGetTPtr(rtwdemo_basicsc_M));
}

```

Alternatively, to configure the System Outputs block, at the command prompt, use these commands:

```

temp.TLCFile = 'custcode';
temp.Location = 'System Outputs Function';
temp.Middle = sprintf(['\nrtwdemo_basicsc_U.input2++;'...
    '\nrtwCAPI_ModelMappingInfo *MMI = '...
    '&(rtmGetDataMapInfo(rtwdemo_basicsc_M).mmi);'...
    '\ntuneFcn(MMI, rtmGetTPtr(rtwdemo_basicsc_M));\n']);
temp.Bottom = sprintf(['\nrtwCAPI_ModelMappingInfo *MMI = '...
    '&(rtmGetDataMapInfo(rtwdemo_basicsc_M).mmi);'...
    '\nlogFcn(MMI, rtmGetTPtr(rtwdemo_basicsc_M));\n']);
set_param('rtwdemo_basicsc/System Outputs', 'RTWdata', temp)

```

Generate code from the model.

```

rtwbuild('rtwdemo_basicsc')

### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc

```

In the code generation report, view the interface file `rtwdemo_basicsc_capi.c`. This file initializes the arrays of structures that you can use to interact with data items through the C API. For example, in the array of structures `rtBlockSignals`, the first structure (index 0) describes the test-pointed output signal of the Gain block in the model.

```

file = fullfile('rtwdemo_basicsc_grt_rtw', 'rtwdemo_basicsc_capi.c');
rtwdemodbtype(file, /* Block output signal information */',...
    /* Individual block tuning',1,0)

```

```

/* Block output signal information */
static const rtwC_API_Signals rtBlockSignals[] = {
    /* addrMapIndex, sysNum, blockPath,
     * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
     */
    { 0, 0, TARGET_STRING("rtwdemo_basicsc/Gain"),
      TARGET_STRING(""), 0, 0, 0, 0, 0 },

    {
        0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
    }
};

```

The fields of the structure, such as `addrMapIndex`, indicate indices into other arrays of structures, such as `rtDataAddrMap`, that describe the characteristics of the signal. These characteristics include the address of the signal data (a pointer to the data), the numeric data type, and the dimensions of the signal.

In the file `rtwdemo_basicsc.c`, view the code algorithm in the model `step` function. The algorithm first executes the custom code that you specified in the System Outputs block.

```

file = fullfile('rtwdemo_basicsc_grt_rtw', 'rtwdemo_basicsc.c');
rtwdemodbtype(file, '/* user code (Output function Body) */', ...
    '/* DataStoreWrite: '<Root>/DSWrite' incorporates:', 1, 0)

/* user code (Output function Body) */

/* System '<Root>' */
{
    rtwdemo_basicsc_U.input2++;
    rtwC_API_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_basicsc_M).mmi);
    tuneFcn(MMI, rtmGetTPtr(rtwdemo_basicsc_M));
}

```

This custom code first perturbs the input signal `input2` by incrementing the value of the signal each time the `step` function executes. The code then uses the built-in macro `rtmGetDataMapInfo` to extract model mapping information from the model data structure `rtwdemo_basicsc_M`. The pointer `MMI` points to the extracted mapping information, which allows the custom functions `tuneFcn` and `logFcn` to access the information contained in the arrays of structures that the C API file `rtwdemo_basicsc_capi.c` defines.

View the custom function `tuneFcn` in the file `myHandCode.c`. This function uses the C API (through the model mapping information `mmi`) and a pointer to the simulation time to print the value of the parameter `K1` at specific times during code execution. When the simulation time reaches 5 seconds, the function changes the parameter value in memory. By using a `switch case` block, the function can access the parameter data whether the data type is `int8` or `double`.

View the code algorithm in the model `step` function again. Near the end of the function, the algorithm executes the custom code that you specified in the System Outputs block. This code calls the custom function `logFcn`.

```
rtwdemodbtype(file, '/* user code (Output function Trailer) */', ...
    '/* Matfile logging */', 1, 0)

/* user code (Output function Trailer) */

/* System '<Root>' */
{
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_bascisc_M).mmi);
    logFcn(MMI, rtmGetTPtr(rtwdemo_bascisc_M));
}
```

View the custom function `logFcn` in the file `myHandCode.c`. The function uses the C API to print the value of the test-pointed signal. The function can access the signal data whether the data type is `single` or `double`.

At the command prompt, run the generated executable `rtwdemo_bascisc.exe`.

```
system('rtwdemo_bascisc')
```

The parameter and signal values appear in the Command Window output.

For more information about data interfaces, including the C API, see “Data Exchange Interfaces”.

See Also

`Simulink.Parameter` | `Simulink.Signal`

Related Examples

- “Default Data Structures in the Generated Code” on page 14-16

- “Configure Block Parameter Tunability for Rapid Prototyping” on page 14-52
- “Control Signals and States in Code by Applying Storage Classes” on page 14-120
- “Exchange Data Between Generated and External Code Using C API” on page 28-2

Default Data Structures in the Generated Code

The generated code creates variables to represent model data such as signals, block parameters, and states. The code generation settings that you choose for a model determine the default scope of each datum. If the code generator applies a global scope to a datum, by default the datum appears as a field of a global data structure rather than a separate global variable. For example, the generated code creates default structures to contain block output signals, tunable parameters, and constant-valued nontunable parameters that the code generator cannot inline.

The table shows the most common global data structures in the generated code. The default name of each structure variable is *model_structname*. *model* is the name of the model. *structname* is the structure name in the table.

Global data structures generated for a standalone model

Structure Name	Data Represented in the Structure
U	Data from root Inport blocks
Y	Data from root Outport blocks
B	Block output signals
ConstB	Block outputs that have constant values
P	Block parameters
DefaultP	Default parameters in the system
ConstP	Constant parameters
DW	Discrete block states
X	Continuous block states
XDot	Derivatives of continuous states at each time step
XDis	Status of enabled subsystems
ZCV	Zero-crossing signals
PrevZCX	Previous zero-crossing signal states
Obj	Used by ERT C++ code generation to refer to referenced model objects

The table shows the most common global data structures generated for atomic subsystems and referenced models. The default name of each structure variable is *model_structname* for referenced models and *model_subsystem_structname* for subsystems.

Global data structures generated for subsystems and referenced models

Structure Name	Data Represented in the Structure
B	Block output signals
ConstB	Block outputs that have constant values
P	Block parameters
DW	Discrete block states
MdlRefDW	Discrete block states in referenced model
X	Continuous states in model reference
XDis	Status of enabled subsystems
ZCV	Zero-crossing signals
RTM	RT_Model structure

If you have an Embedded Coder license, you can control the names of these global structure variables. For more information, see “Global variables” and “System-generated identifiers”.

You can exclude data from appearing in these structures by using:

- Storage classes. For example, you can use storage classes to represent signals, tunable parameters, and states as individual global variables. For more information, see “Control Signals and States in Code by Applying Storage Classes” on page 14-120 and “Block Parameter Representation in the Generated Code” on page 14-43.
- Configuration parameters, such as those on the **Optimization > Signals and Parameters** pane in the Configuration Parameters dialog box. You can adjust these configuration parameters to control the default representation of data. For more information, see “Optimization Pane: Signals and Parameters”.

See Also

“Combine signal/state structures”

Related Examples

- “Signal Representation in Generated Code” on page 14-109
- “Block Parameter Representation in the Generated Code” on page 14-43
- “Access Signal, State, and Parameter Data During Execution” on page 14-3

Use the Real-Time Model Data Structure

The code generator uses the real-time model (RT_MODEL) data structure. This structure is also referred to as the `rtModel` data structure. You can access `rtModel` data by using a set of macros analogous to the `ssSetxxx` and `ssGetxxx` macros that S-functions use to access `SimStruct` data, including noninlined S-functions compiled by the code generator.

You need to use the set of macros `rtmGetxxx` and `rtmSetxxx` to access the real-time model data structure. The `rtModel` is an optimized data structure that replaces `SimStruct` as the top level data structure for a model. The `rtmGetxxx` and `rtmSetxxx` macros are used in the generated code as well as from the `main.c` or `main.cpp` module. If you are customizing `main.c` or `main.cpp` (either a static file or a generated file), you need to use `rtmGetxxx` and `rtmSetxxx` instead of the `ssSetxxx` and `ssGetxxx` macros.

Usage of `rtmGetxxx` and `rtmSetxxx` macros is the same as for the `ssSetxxx` and `ssGetxxx` versions, except that you replace `SimStruct S` by real-time model data structure `rtM`. The following table lists `rtmGetxxx` and `rtmSetxxx` macros that are used in `grt_main.c` and `grt_main.cpp`.

Macros for Accessing the Real-Time Model Data Structure

rtm Macro Syntax	Description
<code>rtmGetdX(rtM)</code>	Get the derivatives of block continuous states
<code>rtmGetOffsetTimePtr(RT_MDL rtM)</code>	Return the pointer to vector that stores sample time offsets of the model associated with <code>rtM</code>
<code>rtmGetNumSampleTimes(RT_MDL rtM)</code>	Get the number of sample times that a block has
<code>rtmGetPerTaskSampleHitsPtr(RT_MDL)</code>	Return a pointer to <code>NumSampleTime × NumSampleTime</code> matrix
<code>rtmGetRTWExtModeInfo(RT_MDL rtM)</code>	Return an external mode information data structure of the model (used internally for external mode)
<code>rtmGetRTWLogInfo(RT_MDL)</code>	Return a data structure used by code generator logging (internal use only)
<code>rtmGetRTWRModelMethodsInfo(RT_MDL)</code>	Return a data structure of real-time model methods information (internal use only)
<code>rtmGetRTWSolverInfo(RT_MDL)</code>	Return data structure containing solver information of the model (internal use only)

rtm Macro Syntax	Description
<code>rtmGetSampleHitPtr(RT_MDL)</code>	Return a pointer to Sample Hit flag vector
<code>rtmGetSampleTime(RT_MDL rtm, int TID)</code>	Get task sample time
<code>rtmGetSampleTimePtr(RT_MDL rtm)</code>	Get pointer to a task sample time
<code>rtmGetSampleTimeTaskIDPtr(RT_MDL rtm)</code>	Get pointer to a task ID
<code>rtmGetSimTimeStep(RT_MDL)</code>	Return simulation step type ID (MINOR_TIME_STEP, MAJOR_TIME_STEP)
<code>rtmGetStepSize(RT_MDL)</code>	Return the fundamental step size of the model
<code>rtmGetT(RT_MDL, t)</code>	Get the current simulation time
<code>rtmSetT(RT_MDL, t)</code>	Set the time of the next sample hit
<code>rtmGetTaskTime(RT_MDL, tid)</code>	Get the current time for the current task
<code>rtmGetTFinal(RT_MDL)</code>	Get the simulation stop time
<code>rtmSetTFinal(RT_MDL, finalT)</code>	Set the simulation stop time
<code>rtmGetTimingData(RT_MDL)</code>	Return a data structure used by timing engine of the model (internal use only)
<code>rtmGetTPtr(RT_MDL)</code>	Return a pointer to the current time
<code>rtmGetTStart(RT_MDL)</code>	Get the simulation start time
<code>rtmIsContinuousTask(rtm)</code>	Determine whether a task is continuous
<code>rtmIsMajorTimeStep(rtm)</code>	Determine whether the simulation is in a major step
<code>rtmIsSampleHit(RT_MDL, tid)</code>	Determine whether the sample time is hit
<code>rtmGetErrorStatus(rtm)</code>	Get the current error status
<code>rtmSetErrorStatus(rtm, val)</code>	Set the current error status
<code>rtmGetErrorStatusPointer(rtm)</code>	Return a pointer to the current error status
<code>rtmGetStopRequested(rtm)</code>	Return whether a stop is requested
<code>rtmGetBlockIO(rtm)</code>	Get the block I/O data structure
<code>rtmSetBlockIO(rtm, val)</code>	Set the block I/O data structure
<code>rtmGetContStates(rtm)</code>	Get the continuous states data structure
<code>rtmSetContStates(rtm, val)</code>	Set the continuous states data structure

rtm Macro Syntax	Description
<code>rtmGetDefaultParam(rtm)</code>	Get the default parameters data structure
<code>rtmSetDefaultParam(rtm, val)</code>	Set the default parameters data structure
<code>rtmGetPrevZCSigState(rtm)</code>	Get the previous zero-crossing signal state data structure
<code>rtmSetPrevZCSigState(rtm, val)</code>	Set the previous zero-crossing signal state data structure
<code>rtmGetRootDWork(rtm)</code>	Get the DWork data structure
<code>rtmSetRootDWork(rtm, val)</code>	Set the DWork data structure
<code>rtmGetU(rtm)</code>	Get the root inputs data structure (when root inputs are passed as part of the model data structure)
<code>rtmSetU(rtm, val)</code>	Set the root inputs data structure (when root inputs are passed as part of the model data structure)
<code>rtmGetY(rtm)</code>	Get the root outputs data structure (when root outputs are passed as part of the model data structure)
<code>rtmSetY(rtm, val)</code>	Set the root outputs data structure (when root outputs are passed as part of the model data structure)

For additional details on usage, see “SimStruct Macros and Functions Listed by Usage”.

Related Examples

- “SimStruct Macros and Functions Listed by Usage”
- “System Target Files and Code Generation Features” on page 17-18

Use Enumerated Data in Generated Code

In this section...

“Enumerated Data Types” on page 14-22

“Specify Integer Data Type for Enumeration” on page 14-22

“Customize Enumerated Data Type” on page 14-24

“Control Enumerated Type Implementation in Generated Code” on page 14-28

“Type Casting for Enumerations” on page 14-29

“Enumerated Type Limitations” on page 14-30

Enumerated Data Types

Enumerated data is data that is restricted to a finite set of values. An *enumerated data type* is a MATLAB class that defines a set of *enumerated values*. Each enumerated value consists of an *enumerated name* and an *underlying integer* which the software uses internally and in generated code. The following is a MATLAB class definition for an enumerated data type named `BasicColors`, which is used in the examples in this section.

```
classdef BasicColors < Simulink.IntEnumType
    enumeration
        Red(0)
        Yellow(1)
        Blue(2)
    end
end
```

For basic information about enumerated data types and their use in Simulink models, see “Use Enumerated Data in Simulink Models”. For information about enumerated data types in Stateflow charts, see “Define Enumerated Data in a Chart”.

Specify Integer Data Type for Enumeration

When you specify a data type for your enumeration, you can:

- Control the size of enumerated data types in the generated code by specifying a superclass.
- Reduce RAM/ROM usage.

- Improve code portability.
- Improve integration with legacy code.

You can specify any of these integer data types:

- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `Simulink.IntEnumType`. Specify values in the range of the signed integer for your hardware platform.

Use a Class Definition in a MATLAB File

To specify an integer data type size, derive your enumeration class from the integer data type.

```
classdef Colors < int8
    enumeration
        Red(0)
        Green(1)
        Blue(2)
    end
end
```

The code generator generates this code:

```
typedef int8_T Colors;

#define Red      ((Colors)0)
#define Green   ((Colors)1)
#define Blue    ((Colors)2)
```

Use the Function `Simulink.defineIntEnumType`

To specify an integer data type size, specify the name-value pair `StorageType` as the integer data type.

```
Simulink.defineIntEnumType('Colors', {'Red', 'Green', 'Blue'}, ...
[0;1;2], 'StorageType', 'int8')
```

The code generator generates this code:

```
typedef int8_T Colors;

#define Red      ((Colors)0)
#define Green   ((Colors)1)
#define Blue    ((Colors)2)
```

Customize Enumerated Data Type

When you generate code from a model that uses enumerated data, you can implement these static methods to customize the behavior of the type during simulation and in generated code:

- `getDefaultValue` — Specifies the default value of the enumerated data type.
- `getDescription` — Specifies a description of the enumerated data type.
- `getHeaderFile` — Specifies a header file where the type is defined for generated code.
- `getDataScope` — Specifies whether generated code exports or imports the enumerated data type definition to or from a separate header file.
- `addClassNameToEnumNames` — Specifies whether the class name becomes a prefix in generated code.

The first of these methods, `getDefaultValue`, is relevant to both simulation and code generation, and is described in “Specify a Default Enumerated Value” in the Simulink documentation. The other methods are relevant only to code generation. To customize the behavior of an enumerated type, include a version of the method in the `methods(Static)` section of the enumeration class definition. If you do not want to customize the type, omit the `methods(Static)` section. The table summarizes the methods and the data to supply for each one.

Static Method	Purpose	Default Value Without Implementing Method	Custom Return Value
<code>getDefaultValue</code>	Specifies the default enumeration member for the class.	First member specified in the enumeration definition	A character vector containing the name of an enumeration member in the class (see “Instantiate Enumerations”).

Static Method	Purpose	Default Value Without Implementing Method	Custom Return Value
<code>getDescription</code>	Specifies a description of the enumeration class.	' '	A character vector containing the description of the type.
<code>getHeaderFile</code>	Specifies the name of a header file. The method <code>getDataScope</code> determines the significance of the file.	' '	A character vector containing the name of the header file that defines the enumerated type. By default, the generated <code>#include</code> directive uses the preprocessor delimiter <code>"</code> instead of <code><</code> and <code>></code> . To generate the directive <code>#include <myTypes.h></code> , specify the custom return value as <code>'<myTypes.h>'</code> .
<code>getDataScope</code>	Specifies whether generated code exports or imports the definition of the enumerated data type. Use the method <code>getHeaderFile</code> to specify the generated or included header file that defines the type.	'Auto'	One of: 'Auto', 'Exported', or 'Imported'.
<code>addClassNameToEnumName</code>	Specifies whether to prefix the class name in generated code.	false	true or false.

Specify a Description

To specify a description for an enumerated data type, include this method in the `methods(Static)` section of the enumeration class:

```
function retVal = getDescription()
% GETDESCRIPTION Optional description of the data type.
    retVal = 'description';
end
```

Substitute a MATLAB character vector for *description*. The generated code that defines the enumerated type includes the specified description.

Import Type Definition in Generated Code

To prevent generated code from defining an enumerated data type, which allows you to provide the definition in an external file, include these methods in the `methods(Static)` section of the enumeration class:

```
function retVal = getHeaderFile()
% GETHEADERFILE Specifies the file that defines this type in generated code.
% The method getDataScope determines the significance of the specified file.
    retVal = 'imported_enum_type.h';
end

function retVal = getDataScope()
% GETDATASCOPE Specifies whether generated code imports or exports this type.
% Return one of:
% 'Auto':    define type in model_types.h, or import if header file specified
% 'Exported': define type in a generated header file
% 'Imported': import type definition from specified header file
% If you do not define this method, DataScope is 'Auto' by default.
    retVal = 'Imported';
end
```

Instead of defining the type in `model_types.h`, which is the default behavior, generated code imports the definition from the specified header file using a `#include` statement like:

```
#include "imported_enum_type.h"
```

Generating code does not create the imported header file. You must provide the header file, using the file name specified by the method `getHeaderFile`, that defines the enumerated data type.

Export Type Definition in Generated Code

To generate a separate header file that defines an enumerated data type, include these methods in the `methods(Static)` section of the enumeration class:

```

function retVal = getDataScope()
    % GETDATASCOPE Specifies whether generated code imports or exports this type.
    % Return one of:
    % 'Auto':    define type in model_types.h, or import if header file specified
    % 'Exported': define type in a generated header file
    % 'Imported': import type definition from specified header file
    % If you do not define this method, DataScope is 'Auto' by default.
    retVal = 'Exported';
end

function retVal = getHeaderFile()
    % GETHEADERFILE Specifies the file that defines this type in generated code.
    % The method getDataScope determines the significance of the specified file.
    retVal = 'exported_enum_type.h';
end

```

Generated code exports the enumerated type definition to the generated header file `exported_enum_type.h`.

Add Prefixes To Class Names

By default, enumerated values in generated code have the same names that they have in the enumeration class definition. Alternatively, your code can prefix every enumerated value in an enumeration class with the name of the class. You can use this technique to prevent identifier conflicts or to improve the readability of the code. To specify class name prefixing, include this method in the `methods(Static)` section of an enumeration class:

```

function retVal = addClassNameToEnumNames()
    % ADDCLASSNAMETOENUMNAMES Specifies whether to add the class name
    % as a prefix to enumeration member names in generated code.
    % Return true or false.
    % If you do not define this method, no prefix is added.
    retVal = true;
end

```

Specify the return value as `true` to enable class name prefixing or as `false` to suppress prefixing. If you specify `true`, each enumerated value in the class appears in generated code as `EnumTypeName_EnumName`. For the example enumeration class `BasicColors` in “Enumerated Data Types” on page 14-22, the data type definition in generated code might look like this:

```

#ifndef _DEFINED_TYPEDEF_FOR_BasicColors_
#define _DEFINED_TYPEDEF_FOR_BasicColors_

typedef enum {
    BasicColors_Red = 0,          /* Default value */
    BasicColors_Yellow = 1,
    BasicColors_Blue = 2,
} BasicColors;

```

```
#endif
```

The enumeration class name `BasicColors` appears as a prefix for each of the enumerated names.

Control Enumerated Type Implementation in Generated Code

Suppose that you define an enumerated type `BasicColors`. You can specify that the generated code implement the type definition using:

- An `enum` block. The native integer type of your hardware is the underlying integer type for the enumeration members.
- A `typedef` statement and a series of `#define` macros. The `typedef` statement bases the enumerated type name on a specific integer data type, such as `int8`. The macros associate the enumeration members with the underlying integer values.

Implement Enumerated Type Using `enum` Block

To implement the type definition using an `enum` block:

- In Simulink, define the enumerated type using a `classdef` block in a script file. Derive the enumeration from the type `Simulink.IntEnumType`.
- Alternatively, use the function `Simulink.defineIntEnumType`. Do not specify the property `StorageType`.

When you generate code, the type definition appears in an `enum` block.

```
#ifndef _DEFINED_TYPEDEF_FOR_BasicColors_  
#define _DEFINED_TYPEDEF_FOR_BasicColors_  
  
typedef enum {  
    Red = 0,           /* Default value */  
    Yellow,  
    Blue,  
} BasicColors;  
  
#endif
```

Implement Enumerated Type Using a Specific Integer Type

To implement the type definition using a `typedef` statement and `#define` macros:

- In Simulink, define the enumerated type using a `classdef` block in a script file. Derive the enumeration from a specific integer type such as `int8`.
- Alternatively, use the function `Simulink.defineIntEnumType`. Specify the property `StorageType` using a specific integer type such as `int8`.

When you generate code, the type definition appears as a `typedef` statement and a series of `#define` macros.

```
#ifndef _DEFINED_TYPEDEF_FOR_BasicColors_
#define _DEFINED_TYPEDEF_FOR_BasicColors_

typedef int8_T BasicColors;

#define Red ((BasicColors)0)          /* Default value */
#define Yellow ((BasicColors)1)
#define Blue ((BasicColors)2)

#endif
```

By default, the generated file `model_types.h` contains enumerated type definitions.

Type Casting for Enumerations

Safe Casting

A Simulink Data Type Conversion block accepts a signal of integer type. The block converts the input to one of the underlying values of an enumerated type.

If the input value does not match any of the underlying values of the enumerated type values, Simulink inserts a safe cast to replace the input value with the enumerated type default value.

Enable and Disable Safe Casting

You can enable or disable safe casting for enumerations during code generation for a Simulink Data Type Conversion block or a Stateflow block.

To control safe casting, enable or disable the **Saturate on integer overflow** block parameter. The parameter works as follows:

- **Enabled:** Simulink replaces a nonmatching input value with the default value of the enumerated values during simulation. The software generates a safe cast function during code generation.

- **Disabled:** For a nonmatching input value, Simulink generates an error during simulation. The software omits the safe cast function during code generation. In this case, the code is more efficient. However, the code may be more vulnerable to run-time errors.

Safe Cast Function in Generated Code

This example shows how the safe cast function `int32_T ET08_safe_cast_to_BasicColors` for the enumeration `BasicColors` appears in generated code when generated for 32-bit hardware.

```
static int32_T ET08_safe_cast_to_BasicColors(int32_T input)
{
    int32_T output;
    /* Initialize output value to default value for BasicColors (Red) */
    output = 0;
    if ((input >= 0) && (input <= 2)) {
        /* Set output value to input value if it is a member of BasicColors */
        output = input;
    }
    return output;
}
```

Through this function, the enumerated type's default value is used if the input value does not match one of underlying values of the enumerated type's values.

If the block's **Saturate on integer overflow** parameter is disabled, this function does not appear in generated code.

Enumerated Type Limitations

- Generated code does not support logging enumerated data.

See Also

`enumeration` | `Simulink.data.getEnumTypeInfo` |
`Simulink.defineIntEnumType`

Related Examples

- “Use Enumerated Data in Simulink Models”
- “Simulink Enumerations”

Data Stores in Generated Code

In this section...

“About Data Stores” on page 14-31

“Generate Code for Data Store Memory Blocks” on page 14-31

“Storage Classes for Data Store Memory Blocks” on page 14-32

“Data Store Buffering in Generated Code” on page 14-34

About Data Stores

A data store contains data that is accessible in a model hierarchy at or below the level in which the data store is defined. Data stores can allow subsystems and referenced models to share data without having to use I/O ports to pass the data from level to level. See “Data Stores with Data Store Memory Blocks” for information about data stores in Simulink. This section provides additional information about data store code generation.

Generate Code for Data Store Memory Blocks

To control the code generated for a Data Store Memory block, apply a storage class to the data store. You can associate a Data Store Memory block with a signal object that you store in a workspace or data dictionary, and control code generation for the block by applying the storage class to the object:

- 1 Instantiate the desired signal object.
- 2 Set the object's `CoderInfo.StorageClass` property to indicate the desired storage class.
- 3 Open the block dialog box for the Data Store Memory block that you want to associate with the signal object.
- 4 Enter the name of the signal object in the **Data store name** field.
- 5 Select **Data store name must resolve to Simulink signal object**.
- 6 *Do not* set the storage class field to a value other than `Auto` (the default).
- 7 Click **OK** or **Apply**.

Note When a Data Store Memory block is associated with a signal object, the mapping between the **Data store name** and the signal object name must be one-

to-one. If two or more identically named entities map to the same signal object, the name conflict is flagged as an error at code generation time. See “Resolve Conflicts in Configuration of Signal Objects” on page 14-128 for more information.

Storage Classes for Data Store Memory Blocks

You can control how Data Store Memory blocks in your model are stored and represented in the generated code by assigning storage classes and type qualifiers. You do this in almost exactly the same way you assign storage classes and type qualifiers for block states.

Data Store Memory blocks, like block states, have `Auto` storage class by default, and their memory is stored within the `DWork` vector. The symbolic name of the storage location is based on the data store name.

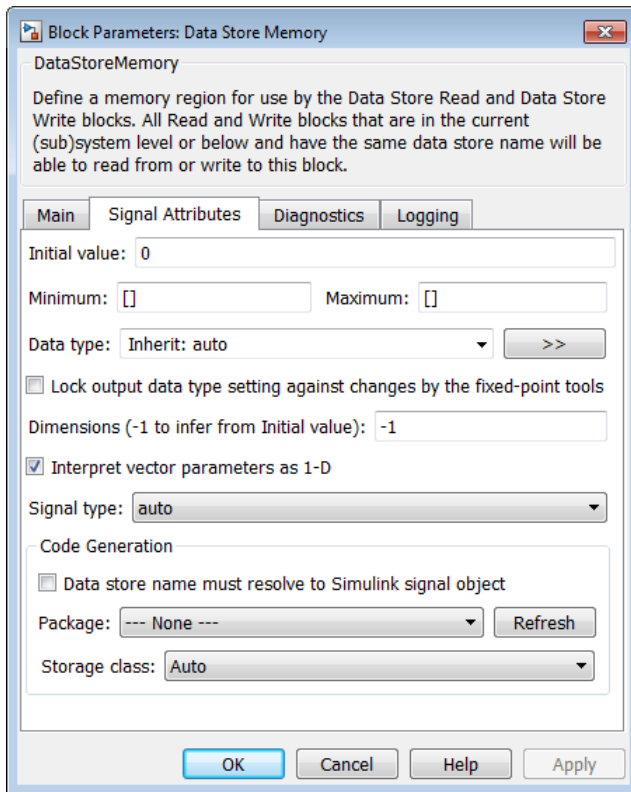
You can generate code from multiple Data Store Memory blocks that have the same data store name, subject to the following restriction: *at most one* of the identically named blocks can have a storage class other than `Auto`. An error is reported if this condition is not met.

For blocks with `Auto` storage class, the code generator produces a unique symbolic name for each block to avoid name clashes. For Data Store Memory blocks with storage classes other than `Auto`, the generated code uses the data store name as the symbol.

In the following model, a Data Store Write block writes to memory declared by the Data Store Memory block `myData`:



To control the storage declaration for a Data Store Memory block, use the **Code Generation > Signal object class** and **Code Generation > Storage class** drop-down lists of the Data Store Memory block dialog box. Set **Signal object class** to `Simulink.Signal` (the default), and choose a storage class from the **Storage class** drop-down list. The next figure shows the Data Store Memory block dialog box for the preceding model.



Data Store Memory blocks are nonvirtual because code is generated for their initialization in `.c` and `.cpp` files and their declarations in header files. The following table shows how the code generated for the Data Store Memory block in the preceding model differs for different settings of **Code Generation > Storage class**. The table gives the variable declarations and `MdlOutputs` code generated for the `myData` block.

Storage Class	Declaration	Code
Auto or SimulinkGlobal	In <code>model.h</code> <pre>typedef struct D_Work_tag { real_T myData; } D_Work;</pre>	<pre>model_DWork.myData = rtb_SineWave;</pre>

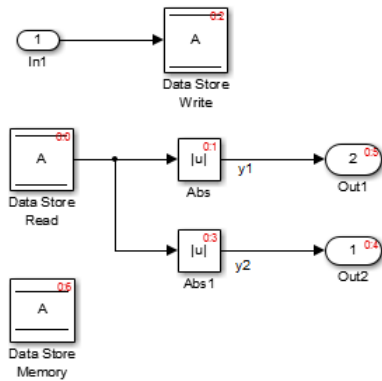
Storage Class	Declaration	Code
	In <i>model.c</i> or <i>model.cpp</i> /* Block states (auto storage) */ D_Work <i>model_DWork</i> ;	
ExportedGlobal	In <i>model.c</i> or <i>model.cpp</i> /* Exported block states */ real_T <i>myData</i> ; In <i>model.h</i> extern real_T <i>myData</i> ;	<i>myData</i> = <i>rtb_SineWave</i> ;
ImportedExtern	In <i>model_private.h</i> extern real_T <i>myData</i> ;	<i>myData</i> = <i>rtb_SineWave</i> ;
ImportedExternPointer	In <i>model_private.h</i> extern real_T * <i>myData</i> ;	(* <i>myData</i>) = <i>rtb_SineWave</i> ;

For information about applying storage classes, see “Control Signals and States in Code by Applying Storage Classes” on page 14-120.

Data Store Buffering in Generated Code

A **Data Store Read** block is a nonvirtual block that copies the value of the data store to its output buffer when it executes. Since the value is buffered, downstream blocks connected to the output of the data store read utilize the same value, even if a **Data Store Write** block updates the data store in between execution of two of the downstream blocks.

The next figure shows a model that uses blocks whose priorities have been modified to achieve a particular order of execution:



```

/* local block i/o variables */
real_T rtb_DataStoreRead;

/* DataStoreRead: '<Root>/Data Store Read' */
rtb_DataStoreRead = A; Buffer the value of A

/* Abs: '<Root>/Abs1' incorporates:
 * DataStoreRead: '<Root>/Data Store Read'
 */
y1 = fabs(A); Use A (whose value equals
the buffered value at this point

/* DataStoreWrite: '<Root>/Data Store Write' incorporates:
 * Inport: '<Root>/In1'
 */
A = u1; Update the value of A

/* Abs: '<Root>/Abs' */
y2 = fabs(rtb_DataStoreRead); Consistently use the same buffered
value as before the update to A

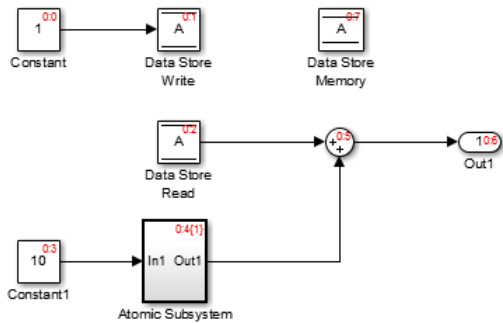
```

The following execution order applies:

- 1 The block Data Store Read buffers the current value of the data store A at its output.
- 2 The block Abs1 uses the buffered output of Data Store Read.
- 3 The block Data Store Write updates the data store.
- 4 The block Abs uses the buffered output of Data Store Read.

Because the output of Data Store Read is a buffer, both Abs and Abs1 use the same value: the value of the data store at the time that Data Store Read executes.

The next figure shows another example:



```

real_T rtb_DataStoreRead;

/* DataStoreWrite: '<Root>/Data Store Write' incorporates:
 * Constant: '<Root>/Constant'
 */
A = DoBufferDSMRead2_P.Constant_Value;

/* DataStoreRead: '<Root>/Data Store Read' */
rtb_DataStoreRead = A; Buffer the value of A

/* Outputs for atomic SubSystem: '<Root>/Atomic Subsystem' */
DoBufferDSMRead_AtomicSubsystem(); We don't do a global analysis to
detect if this function writes to A

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

/* Output: '<Root>/Out1' incorporates:
 * Sum: '<Root>/Sum'
 */
DoBufferDSMRead2_Y.Out1 = rtb_DataStoreRead + DoBufferDSMRead2_B.Abs; Use the buffered value of A

```

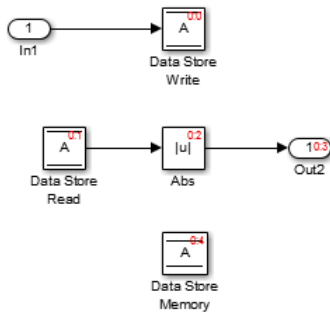
In this example, the following execution order applies:

- 1 The block Data Store Read buffers the current value of the data store A at its output.
- 2 Atomic Subsystem executes.
- 3 The Sum block adds the output of Atomic Subsystem to the output of Data Store Read.

Simulink assumes that Atomic Subsystem might update the data store, so Simulink buffers the data store. Atomic Subsystem executes after Data Store Read buffers its

output, and the buffer provides a way for the Sum block to use the value of the data store as it was when Data Store Read executed.

In some cases, the code generator determines that it can optimize away the output buffer for a Data Store Read block, and the generated code refers to the data store directly, rather than a buffered value of it. The next figure shows an example:



```

/* Model step function */
void DONTbufferDSMRead_step(void)
{
    /* DataStoreWrite: '<Root>/Data Store Write' incorporates:
     * Inport: '<Root>/In1'
     */
    A = u1;

    /* Abs: '<Root>/Abs' incorporates:
     * DataStoreRead: '<Root>/Data Store Read'
     */
    y2 = fabs(A);
}

```

In the generated code, the argument of the `fabs()` function is the data store `A` rather than a buffered value.

Related Examples

- “Control Signals and States in Code by Applying Storage Classes” on page 14-120
- “Structures in Generated Code Using Data Stores” on page 14-38
- “When to Use a Data Store”

Structures in Generated Code Using Data Stores

If you use more than one data store to provide global access to multiple signals in generated code, you can combine the signals into a single structure variable by using one data store. This combination of signal data can help you integrate the code generated from a model with other existing code that requires the data in a structure format.

This example shows how to store several model signals in a structure in generated code using a single data store. To store multiple signals in a data store, you configure the data store to accept a composite signal, such as a nonvirtual bus signal or an array of nonvirtual bus signals.

Explore Example Model

- 1 Open the example model `ex_bus_struct_in_code`.

The model contains three subsystems that perform calculations on the inputs from the top level of the model. In each subsystem, a Data Store Memory block stores an intermediate calculated signal.

- 2 Generate code with the model. In the code generation report, view the file `ex_bus_struct_in_code.c`. The code defines a global variable for each data store.

```
real_T BioBTURate;  
real_T CoalBTURate;  
real_T GasBTURate;
```

Suppose that you want to integrate code generated from the example model with other existing code. Suppose also that the existing code requires access to all of the data from the three data stores in a single structure variable. You can use a data store to assemble all of the target data in a structure in generated code.

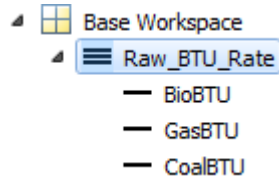
Configure Data Store

Configure a data store to contain multiple signals by creating a bus type to use as the data type of the data store. Define the bus type using the same hierarchy of elements as the structure that you want to appear in generated code.

- 1 Open the Bus Editor tool.

```
buseditor
```

- 2 Define a new bus type `Raw_BTU_Rate` with one element for each of the three target signals. Name the elements `BioBTU`, `GasBTU`, and `CoalBTU`.

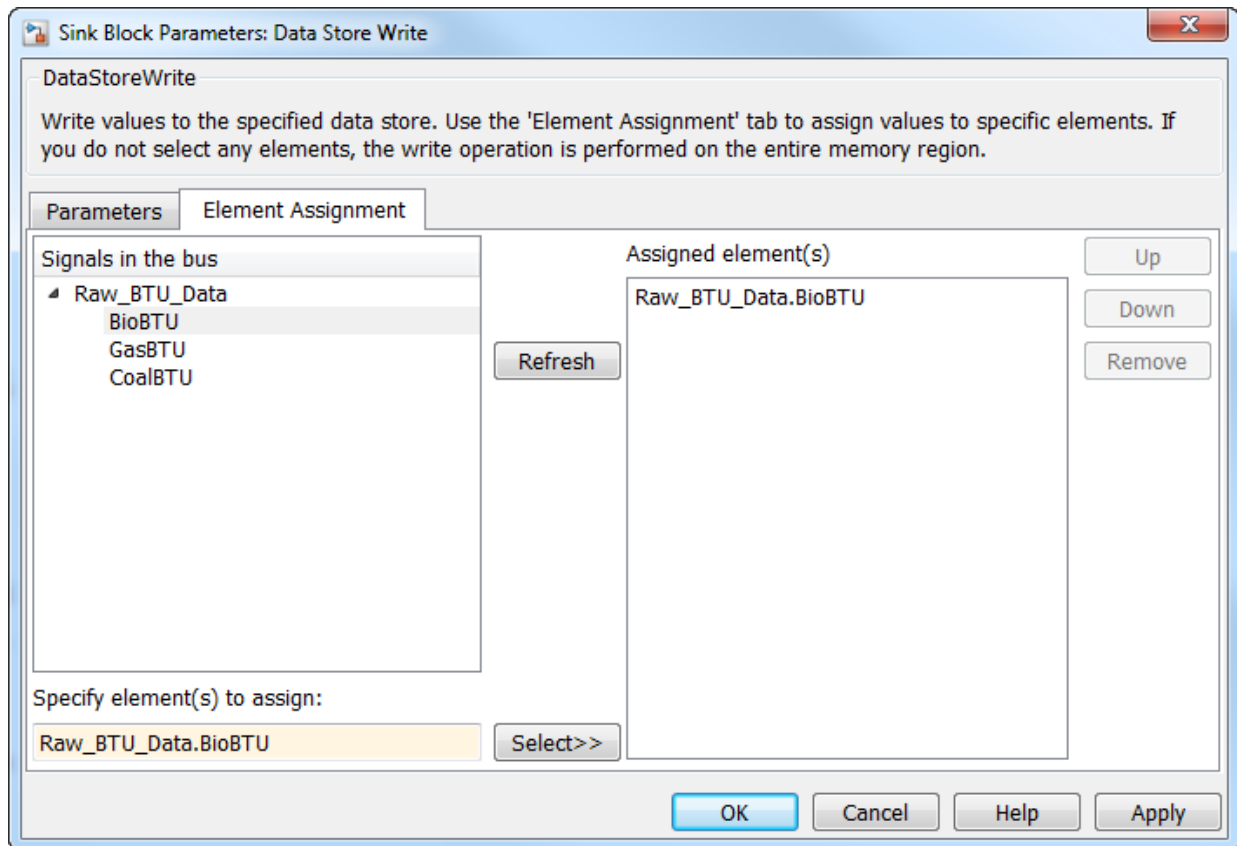


- 3 At the top level of the example model, add a Data Store Memory block.
- 4 In the block dialog box, set **Data store name** to `Raw_BTU_Data`. Click **Apply**.
- 5 On the **Signal Attributes** tab, set **Data type** to `Bus: Raw_BTU_Rate`.
- 6 Under **Code Generation**, set **Storage class** to `ExportedGlobal`.

Write to Data Store Elements

To write to a specific element of a data store, use a Data Store Write block. On the **Element Assignment** tab in the dialog box, you can specify to write to a single element, a collection of elements, or the entire contents of a data store.

- 1 Open the **Biomass Calc** subsystem.
- 2 Delete the Data Store Memory block `BioBTURate`.
- 3 In the block dialog box for the Data Store Write block, set **Data store name** to `Raw_BTU_Data`.
- 4 On the **Element Assignment** tab, under **Signals in the bus**, expand the contents of the data store `Raw_BTU_Data`. Click the element `BioBTU`, and then click **Select**. Click **OK**.



- 5 Modify the **Gas Calc** and **Coal Calc** subsystems similarly.
 - Delete the Data Store Memory block in each subsystem.
 - In each Data Store Write block dialog box, set **Data store name** to `Raw_BTU_Data`.
 - In the **Gas Calc** subsystem, use the Data Store Write block to write to the data store element `GasBTU`. In the **Coal Calc** subsystem, write to the element `CoalBTU`.

Generate Code with Data Store Structure

- 1 Generate code for the example model.
- 2 In the code generation report, view the file `ex_bus_struct_in_code_types.h`. The code defines a structure that corresponds to the bus type `Raw_BTU_Rate`.

```
typedef struct {  
    real_T BioBTU;  
    real_T GasBTU;  
    real_T CoalBTU;  
} Raw_BTU_Rate;
```

- 3 View the file `ex_bus_struct_in_code.c`. The code represents the data store with a global variable `Raw_BTU_Data` of the structure type `Raw_BTU_Rate`. In the model step function, the code assigns the data from the calculated signals to the fields of the global variable `Raw_BTU_Data`.

See Also

[Data Store Read](#) | [Data Store Memory](#) | [Data Store Write](#) | [Simulink.Bus](#)

Related Examples

- “Data Stores with Buses and Arrays of Buses”
- “Access Data Stores with Simulink Blocks”
- “When to Use a Data Store”
- “About Data Stores” on page 14-31

Specify Single Precision

When you want code that uses only single precision, such as when you are targeting a single-precision processor, in the Configuration Parameters dialog box, on the **Optimization** pane, set the **Default for underspecified data type** parameter to **single**. This setting prevents the introduction of **double** data types. When you do not specify a data type explicitly, both simulation and the code generator use **single**.

For more information, see “Default for underspecified data type”.

Related Examples

- “Validate a Floating-Point Embedded Model”
- “Data Type Replacement”
- “About Data Types in Simulink”

Block Parameter Representation in the Generated Code

Blocks have numeric parameters that determine how they calculate output values. For example, a Gain block has a **Gain** parameter. A Discrete Transfer Fcn has multiple parameters that represent the coefficients.

When you generate code from a model, block parameters can appear in the code as inlined numbers, formal parameters of functions, or global variables. Control parameter representation so that you can tune parameter values during algorithm execution, integrate the generated code with your handwritten code, and improve execution performance of the generated code.

For information about setting block parameter values in a model, see “Set Block Parameter Values”.

Default Parameter Representation

You can control whether, by default, block parameters appear in the generated code as:

- Tunable fields of a global structure that contains parameter data. For example, if the value of the **Gain** parameter of a Gain block is **5.2**, the generated code algorithm can appear as `input = myModel_P.myGainBlock_Gain * output;` where the field `myModel_P.myGainBlock_Gain` is initialized to **5.2**.

Use this configuration to enable you to change parameter data during execution for experimentation and rapid prototyping.

- Inlined numeric constants whose values you cannot change during execution. For example, the generated code algorithm can appear as `input = 5.2 * output;`.

Use this configuration to optimize the generated code for production. See “Inline Numeric Values of Block Parameters” on page 31-47.

Configure Parameters as Tunable by Default

When you set **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** (see “Default parameter behavior”) to **Tunable** (the default when you use a GRT-based system target file such as `grt.tlc`), the generated code allocates memory to represent block parameters. Therefore, you can tune the block parameter values during code execution.

If you use numeric expressions, such as 3.57 or $5/2$, to specify block parameter values in the model, each block parameter appears in the generated code as a tunable field of the global parameters structure (for example, `model_P`). The code generator initializes each field by using the corresponding block parameter value from the model.

Workspace variables are variables that you use to specify block parameter values in a model. Workspace variables include numeric MATLAB variables and `Simulink.Parameter` objects that you store in a workspace, such as the base workspace, or in a data dictionary.

When you set **Default parameter behavior** to **Tunable**, workspace variables that use the storage class `Auto` appear in the generated code as tunable fields of the global parameters structure. If you use such a variable to specify multiple block parameter values, the variable appears as a single field of the global parameters structure. The code does not create multiple fields to represent the block parameters. Therefore, tuning the field value during code execution changes the mathematical behavior of the model in the same way as tuning the value of the MATLAB variable or parameter object during simulation.

For more information about the default global structures in the generated code that store signal, state, and parameter data, see “Default Data Structures in the Generated Code” on page 14-16.

To configure block parameter tunability for rapid prototyping, see “Configure Block Parameter Tunability for Rapid Prototyping” on page 14-52.

Configure Parameters as Inlined by Default

When you set **Default parameter behavior** to **Inlined** (the default when you use an ERT-based system target file such as `ert.tlc`), the generated code algorithm inlines the numeric values of block parameters. Therefore, you cannot tune the block parameters during code execution. Workspace variables that use the storage class `Auto`, such as numeric MATLAB variables, also appear in the generated code as inlined constants.

However, the code cannot represent some parameters, such as arrays, as inlined constants. These parameters appear as fields of a global structure that contains constant-valued, nontunable parameters. The structure uses the `const` type qualifier.

The code generator pools equivalent parameter values into a single field of the `const` structure instead of creating multiple fields. For example, if two Gain blocks use the nonscalar value `[2 5 7]` for the value of the **Gain** parameter, the code generator creates

a single structure field to store that value, and shares the field between the lines of code for each block. This pooling reduces the memory consumption of the structure and facilitates code reuse (see “Shared Constant Parameters for Code Reuse” on page 6-24).

If you have an Embedded Coder license, you can generate scalar inlined parameters as macros instead of literal numbers to improve code readability. See “Generate scalar inlined parameters as”.

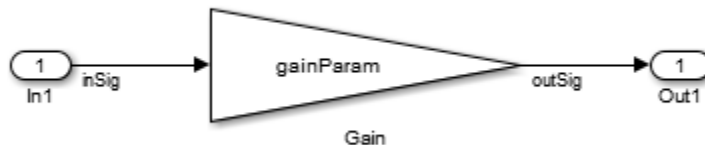
Override Default Parameter Behavior by Creating Global Variables in the Generated Code

You can control the code generated for a block parameter by applying a *storage class* or custom storage class to a `Simulink.Parameter` object. Use the parameter object to set the value of one or more block parameters in a model. By using this technique, you override the setting of **Default parameter behavior** for individual block parameters. For example, you can use this technique to import or export the corresponding generated variable to or from the generated code.

Choose from these built-in storage classes:

- **Auto**: The default storage class. Control the parameter tunability in the generated code by setting the model configuration parameter **Default parameter behavior**.
- **SimulinkGlobal**: Store the parameter object as a field of the generated global parameters structure, `model_P`, as if you set **Default parameter behavior** to **Tunable**. You can tune the parameter during code execution.
- **ExportedGlobal**: Export the declaration and definition of the parameter from the generated code as an individual global variable. You can tune the variable value during execution and use it in your custom code.
- **ImportedExtern**: Import the definition of the parameter from your custom code.
- **ImportedExternPointer**: Import a pointer to the parameter from your custom code.

For each of the storage classes, the table shows the declaration and the code generated for the workspace variable, `gainParam`, in the example model, `param_examp`. The numeric value of `gainParam` is `15.23`.



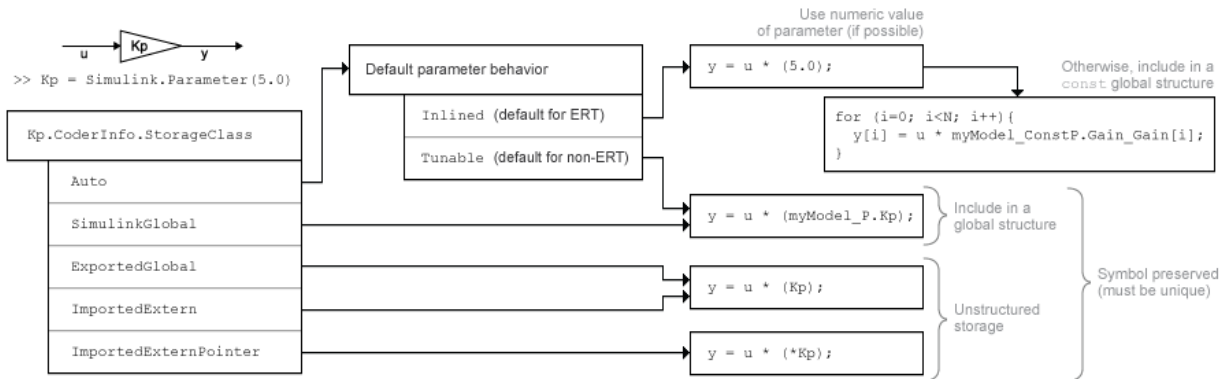
Storage Class	Declaration	Definition	Algorithmic Code
Auto (with Default parameter behavior set to Inlined)	None, because the parameter is inlined	None, because the parameter is inlined	<code>outSig = 15.23 * inSig;</code>
SimulinkGlobal	In <i>model.h</i> <pre>struct P_param_examp_T_ { real_T gainParam; }; extern P_param_examp_T param_examp_P;</pre>	In <i>model_data.c</i> <pre>P_param_examp_T param_examp_P = 15.23 };</pre>	<code>outSig = param_examp_P.gainParam * inSig;</code>
ExportedGlobal	In <i>model.h</i> <pre>extern real_T gainParam;</pre>	In <i>model.c</i> <pre>real_T gainParam = 15.23;</pre>	<code>outSig = gainParam * inSig;</code>
ImportedExported	In <i>model_private.h</i> <pre>extern real_T gainParam;</pre>	None, because the parameter is imported	<code>outSig = gainParam * inSig;</code>
ImportedExported	In <i>model_private.h</i> <pre>extern real_T *gainParam;</pre>	None, because the parameter is imported	<code>outSig = *gainParam * inSig;</code>

For an example of how to use a storage class to control the code generated for a block parameter, see “Create Tunable Calibration Parameter in the Generated Code” on page 14-56.

If you have an Embedded Coder license, you can use and create custom storage classes for greater control over parameter representation. See “Introduction to Custom Storage Classes”.

Parameter Object Configuration Quick Reference Diagram

This diagram shows the code generation and storage class options that control the representation of parameter objects in the generated code.



Preservation of Expressions

You can specify block parameter values as expressions that use `Simulink.Parameter` objects or workspace variables. For example, you can use the expression `5 * gainParam`. See “Use MATLAB Functions and Custom Functions”.

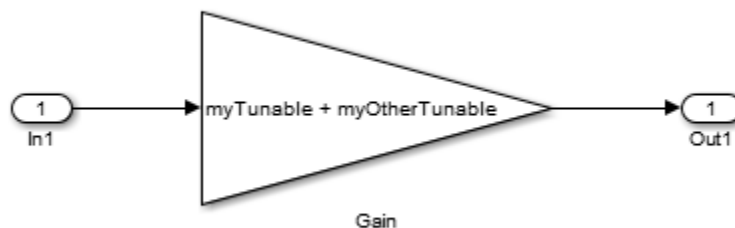
A *tunable workspace variable* is a `Simulink.Parameter` object or workspace variable that appears tunable in the generated code. For example, an object or variable is tunable if you apply a storage class other than `Auto` or if you set **Default parameter behavior** to `Tunable`.

An expression that contains one or more tunable workspace variables, model arguments, or tunable mask parameters is called a *tunable expression*. The expression is tunable because the code generator attempts to preserve the expression in the code. Because the code generator preserves the expression, you can change the values of the parameter data during code execution.

Expression with Tunable Parameter Objects

This example shows how the code generator preserves an expression that you use to set the value of a block parameter.

Consider the model `ex_tunable_expressions`.



The variables `myTunable` and `myOtherTunable` are `Simulink.Parameter` objects that use these property values.

Parameter Object Name	StorageClass Property	Value Property
<code>myTunable</code>	<code>ExportedGlobal</code>	<code>0.75</code>
<code>myOtherTunable</code>	<code>ExportedGlobal</code>	<code>2.0</code>

The generated code represents the tunable parameter objects `myTunable` and `myOtherTunable` as individual global variables. The algorithm preserves the expression `myTunable + myOtherTunable`.

```
ex_tunable_expressions_Y.Out1 = (myTunable + myOtherTunable) *
ex_tunable_expressions_U.In1;
```

Loss of Parameter Tunability

A block parameter, MATLAB variable, or `Simulink.Parameter` object is tunable if it appears in the generated code as data stored in memory, such as a global variable. For example, when you apply the storage class `ExportedGlobal` to a parameter object, the parameter object appears tunable in the generated code. When you set **Default parameter behavior** to `Tunable`, MATLAB variables and parameter objects appear tunable in the generated code. By definition, model arguments also appear tunable.

Under certain conditions, the code generator cannot maintain tunability of a parameter, variable, object, or expression. In this case, the code generator inlines the numeric value, preventing you from changing the value during code execution.

To detect these conditions in your model, set the model configuration parameter **Detect loss of tunability** (see “Detect loss of tunability”) to warning or error.

Tunable Expression Limitations

The code generator reduces certain expressions to an inlined numeric value in the generated code. For example, in “Expression with Tunable Parameter Objects” on page 14-47, you used the expression `myTunable + myOtherTunable` to set the value of a block parameter. If you instead use the expression `myTunable ^ myOtherTunable`, the code generator:

- 1 Evaluates the numeric value of the expression by using the values of the parameter objects. In this case, the expression value is $\text{myTunable}^{\text{myOtherTunable}} = 0.75^{2.0} = 0.5625$.
- 2 Inlines the expression value in the generated code algorithm. In this case, the calculation appears in the code as `ex_tunable_expressions_Y.Out1 = 0.5625 * ex_tunable_expressions_U.In1;`

The code generator removes the tunability of `myTunable` and `myOtherTunable`.

To avoid loss of tunability due to unsupported expressions, observe these guidelines:

- Expressions involving complex (i) workspace variables or parameter objects are not supported.
- Certain operators and functions cause the code generator to reduce expressions and remove tunability. To determine whether an operator or function causes loss of tunability, use the information in this table.

Category	Operators or Functions
1	+ - .* ./ < > <= >= == ~= &
2	* /
3	abs, acos, asin, atan, atan2, boolean, ceil, cos, cosh, exp, floor, log, log10, sign, sin, sinh, sqrt, tan, tanh, single, int8, int16, int32, uint8, uint16, uint32
4	: .^ ^ [] {} . \ .\ ' .' , ;

- Use operators from category 1 without loss of tunability.
- Use operators from category 2 in expressions as long as at least one operand is a scalar. For example, scalar/scalar and scalar/matrix operand combinations are supported, but matrix/matrix combinations are not supported.

- You can use tunable workspace variables as arguments for the functions in category 3. If you use other functions, the code generator removes the tunability of the arguments.
- The operators in category 4 are not supported.
- The Fcn and If blocks do not support tunable expressions for code generation or in referenced models.
- You can specify any data type for the `Simulink.Parameter` objects or workspace variables that makeup expressions. As long as the data type of these variables and objects and the data type of the corresponding block parameters are the same or `double`, the code generator can preserve tunability.

Linear Block Parameter Tunability

These blocks have a `Realization` parameter that affects the tunability of their numeric parameters:

- Transfer Fcn
- State-Space
- Discrete State-Space

To set the `Realization` parameter, you must use the command prompt:

```
set_param(gcf, 'Realization', 'auto')
```

For the `Realization` parameter, you can choose these options:

- **general**: The block's numeric parameters appear tunable in the generated code.
- **sparse**: The generated code represents the block's parameters as transformed values that increase efficiency. The parameters are not tunable.
- **auto**: The default. If one or more of the block's parameters are tunable (for example, because you use a tunable parameter object to set a parameter value), then the block uses the **general** realization. Otherwise, the block uses the **sparse** realization.

To tune the parameter values of one of these blocks during an external mode simulation, the block must use the **general** realization.

See Also

`Simulink.Parameter`

Related Examples

- “Configure Block Parameter Tunability for Rapid Prototyping” on page 14-52
- “Exchange and Reuse Parameter Data Between Generated Code and Existing Code”
- “Switch Between Sets of Parameter Values During Simulation and Code Execution” on page 14-100
- “Set Block Parameter Values”
- “Inline Numeric Values of Block Parameters” on page 31-47
- “Default Data Structures in the Generated Code” on page 14-16

Configure Block Parameter Tunability for Rapid Prototyping

As you iteratively develop a model, to experiment with block parameter values (for example, the **Gain** parameter of a Gain block), you can *tune* the values during simulation or execution of the generated code. You can then observe the effect on signal values, and base your design decisions on analysis of these outputs. To access parameter data, you can configure the generated code to store the data in addressable global memory.

Optimization settings can make the generated code more efficient by inlining the numeric values of block parameters. To generate code that instead allocates addressable memory for parameters, you can disable the optimizations for all block parameters. Regardless of the settings that you use for the optimizations, you can also specify code generation settings for individual block parameters.

Goal	Considerations and More Information
Configure parameters as tunable by default	<p>To prevent block parameter inlining in the generated code, and instead store parameter values in global memory, set the model configuration parameter Default parameter behavior to Tunable. Each block parameter appears in the generated code as a field of a global structure.</p> <p>For more information about Default parameter behavior, see “Default parameter behavior”. For an example, see “Access Signal, State, and Parameter Data During Execution”.</p>
Create separate global variables	<p>When you set Default parameter behavior to Tunable, block parameters appear in the generated code as fields of a structure whose name you cannot explicitly specify. To instead store parameter data in separate global variables whose names, file placement, and other characteristics you can control, apply storage classes to <code>Simulink.Parameter</code> objects. See “Create Tunable Calibration Parameter in the Generated Code” on page 14-56.</p>
Tune parameters during external mode simulation	<p>When you generate code and an external executable from a model, you can simulate the model in external mode to communicate with the running executable. You can tune parameters and monitor signals during the simulation. However, in this simulation mode, tunability limitations that apply to code generation also apply to the simulation. For information about the code generation limitations, see “Loss of Parameter Tunability” on page 14-48.</p>

Goal	Considerations and More Information
	For information about external mode, see “Set Up and Use Host/Target Communication Channel” on page 26-2.
Tune parameters with Simulink Real-Time	If you have Simulink Real-Time, you can tune parameters and monitor signals during execution of your real-time application. Make parameters observable by using storage classes and the model configuration parameter Default parameter behavior . See “Tunable Block Parameters and MATLAB Variables”.

Related Examples

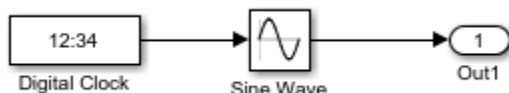
- “Deploy Algorithm Model for Real-Time Rapid Prototyping” on page 23-2
- “Access Signal, State, and Parameter Data During Execution” on page 14-3
- “Block Parameter Representation in the Generated Code” on page 14-43
- “Create Tunable Calibration Parameter in the Generated Code” on page 14-56
- “Switch Between Sets of Parameter Values During Simulation and Code Execution” on page 14-100

Tune Phase Parameter of Sine Wave Block During Code Execution

Under certain conditions, you cannot configure the **Phase** parameter of a Sine Wave block to appear in the generated code as a tunable global variable (for more information, see the block reference page). This example shows how to generate code so that you can tune the phase during execution.

Create the model `ex_phase_tunable`:

```
open_system('ex_phase_tunable')
```



Set **Default parameter behavior** to **Tunable** so that the parameters of the Sine Wave block appear in the generated code as tunable fields of the global parameter structure.

```
set_param('ex_phase_tunable', 'DefaultParameterBehavior', 'Tunable')
```

Generate code from the model.

```
rtwbuild('ex_phase_tunable')
```

```
### Starting build procedure for model: ex_phase_tunable
### Successful completion of code generation for model: ex_phase_tunable
```

In the code generation report, view the file `ex_phase_tunable.c`. The code algorithm in the model `step` function calculates the Sine Wave block output. The parameters of the block, including **Phase**, appear in the code as tunable structure fields.

```
file = fullfile('ex_phase_tunable_grt_rtw', 'ex_phase_tunable.c');
rtwdemodbtype(file, '/ * Outport: '<Root>/Out1' incorporates:', ...
    'ex_phase_tunable_P.SineWave_Bias;', 1, 1)

/* Outport: '<Root>/Out1' incorporates:
 * DigitalClock: '<Root>/Digital Clock'
 * Sin: '<Root>/Sine Wave'
 */
ex_phase_tunable_Y.Out1 = sin(ex_phase_tunable_P.SineWave_Freq *
    (((ex_phase_tunable_M->Timing.clockTick1+
```

```
ex_phase_tunable_M->Timing.clockTickH1* 4294967296.0)) ) +  
ex_phase_tunable_P.SineWave_Phase) * ex_phase_tunable_P.SineWave_Amp +  
ex_phase_tunable_P.SineWave_Bias;
```

During code execution, you can assign new values to the structure field that corresponds to the **Phase** parameter.

Related Examples

- “Configure Block Parameter Tunability for Rapid Prototyping” on page 14-52

Create Tunable Calibration Parameter in the Generated Code

A *calibration parameter* is a value stored in global memory that an algorithm reads for use in calculations but does not write to. Calibration parameters are *tunable* because you can change the stored value during algorithm execution. You create calibration parameters so that you can:

- Determine an optimal parameter value by tuning the parameter and monitoring signal values during execution.
- Efficiently adapt an algorithm to different execution conditions by overwriting the parameter value stored in memory. For example, you can use the same control algorithm for multiple vehicles of different masses by storing different parameter values in each vehicle's engine control unit.

In Simulink, create a `Simulink.Parameter` object to represent a calibration parameter. You use the parameter object to set block parameter values, such as the **Gain** parameter of a Gain block. To control the representation of the parameter object in the generated code, you apply a storage class or custom storage class to the object.

To make block parameters accessible in the generated code by default, for example for rapid prototyping, set **Default parameter behavior** (see “Default parameter behavior”) to **Tunable**. See “Configure Block Parameter Tunability for Rapid Prototyping” on page 14-52.

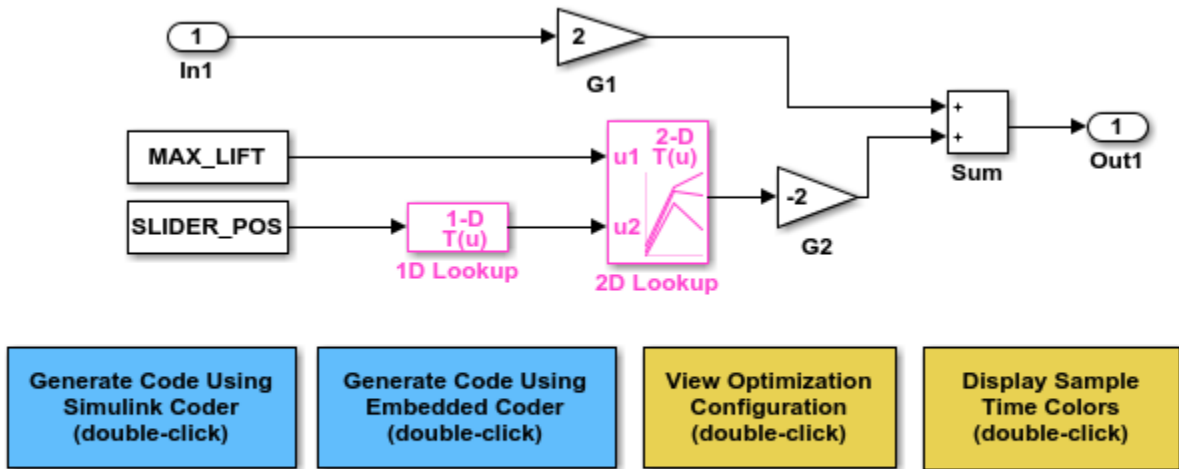
Represent Block Parameter as Tunable Global Variable

This example shows how to create tunable parameter data by representing a block parameter as a global variable in the generated code.

Configure Block Parameter by Using Parameter Object

Open the example model `rtwdemo_paraminline`.

```
rtwdemo_paraminline
```

Copyright 1994-2015 The MathWorks, Inc.

In the G1 block dialog box, change **Gain** from 2 to myGainParam. Click **Apply**.

Right-click myGainParam and select **Create Variable**.

In the Create New Data block dialog box, set **Value** to Simulink.Parameter(2). Click **Create**. A Simulink.Parameter object myGainParam stores the parameter value, 2, in the base workspace.

In the myGainParam dialog box, set **Storage class** to ExportedGlobal and click **OK**. This storage class causes the parameter object to appear in the generated code as a tunable global variable.

Alternatively, to create the parameter object and configure the model, use these commands at the command prompt:

```
set_param('rtwdemo_paraminline/G1','Gain','myGainParam')
myGainParam = Simulink.Parameter(2);
myGainParam.CoderInfo.StorageClass = 'ExportedGlobal';
```

Generate and Inspect Code

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for model: rtwdemo_paraminline
### Successful completion of build procedure for model: rtwdemo_paraminline
```

The generated file `rtwdemo_paraminline.h` contains an `extern` declaration of the global variable `myGainParam`. You can include (`#include`) this header file so that your code can read and write the value of the variable during execution.

```
file = fullfile('rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.h');
rtwdemodbtype(file,'extern real_T myGainParam;', 'extern real_T myGainParam;',1,1)
```

```
extern real_T myGainParam;          /* Variable: myGainParam
```

The file `rtwdemo_paraminline.c` allocates memory for and initializes `myGainParam`.

```
file = fullfile('rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.c');
rtwdemodbtype(file,'/* Exported block parameters */','real_T myGainParam = 2.0;',1,1)
```

```
/* Exported block parameters */
real_T myGainParam = 2.0;          /* Variable: myGainParam
```

The generated code algorithm in the model `step` function uses `myGainParam` for calculations.

```
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function */',1,0)
```

```
/* Model step function */
void rtwdemo_paraminline_step(void)
{
    /* Outport: '<Root>/Out1' incorporates:
     * Gain: '<Root>/G1'
     * Inport: '<Root>/In1'
     * Sum: '<Root>/Sum'
     */
    rtwdemo_paraminline_Y.Out1 = myGainParam * rtwdemo_paraminline_U.In1 + 150.0;
}
```

Configure Accessibility of Signal Data

When you tune the value of a parameter during algorithm execution, you monitor or capture output signal values to analyze the effect of the tuning. To represent signals in

the generated code as accessible data, you can use techniques such as test points and storage classes. See “Signal Representation in Generated Code” on page 14-109.

Programmatic Interfaces for Tuning Parameters

You can configure the generated code to include:

- A C application programming interface (API) for tuning parameters independent of external mode. The generated code includes extra code so that you can write your own code to access parameter values. See “Exchange Data Between Generated and External Code Using C API” on page 28-2.
- A Target Language Compiler API for tuning parameters independently of external mode. See “Parameter Functions” in the Target Language Compiler documentation.

Set Tunable Parameter Minimum and Maximum Values

It is a best practice to specify minimum and maximum values for tunable parameters.

You can specify these minimum and maximum values:

- In the block dialog box that uses the parameter object. Use this technique to store the minimum and maximum information in the model.
- By using the properties of a Simulink.Parameter object that you use to set the parameter value. Use this technique to store the minimum and maximum information outside the model.

For more information, see “Specify Minimum and Maximum Values for Block Parameters”.

Considerations for Other Modeling Goals

Goal	Considerations and More Information
Apply storage type qualifiers such as <code>const</code> and <code>volatile</code>	The storage type qualifier <code>const</code> can protect the integrity of parameter data and enable compilers to optimize machine code. If necessary, you can also use <code>volatile</code> to prevent compilers from eliminating storage for parameter data. If you have Embedded Coder, to generate storage type qualifiers, use custom storage classes. See “Type Qualifiers”.

Goal	Considerations and More Information
Generate ASAP2 description	You can generate an <code>a21</code> file that uses the ASAP2 standard to describe your calibration parameters. For more information, see “Define ASAP2 Information for Parameters and Signals” on page 29-3.
Generate AUTOSAR (<code>arxml</code>) description	If you have an Embedded Coder license, you can generate an <code>arxml</code> file that describes calibration parameters used by models that you configure for the AUTOSAR standard. See “Model AUTOSAR Calibration Parameters and Lookup Tables”.
Store lookup table data for calibration	<p>To store lookup table data for calibration according to the ASAP2 or AUTOSAR standards (for example, <code>STD_AXIS</code>, <code>COM_AXIS</code>, or <code>CURVE</code>), you can use <code>Simulink.LookupTable</code> and <code>Simulink.Breakpoint</code> objects in lookup table blocks. However, some limitations apply. See <code>Simulink.LookupTable</code>.</p> <p>To work around the limitations of <code>Simulink.LookupTable</code> and <code>Simulink.Breakpoint</code> objects, use <code>Simulink.Parameter</code> and <code>AUTOSAR.Parameter</code> objects instead. See “Define ASAP2 Information for Parameters and Signals” on page 29-3 and “Configure <code>STD_AXIS</code> and <code>COM_AXIS</code> Lookup Tables for AUTOSAR Measurement and Calibration”.</p>
Use pragmas to store parameter data in specific memory locations	If you have an Embedded Coder license, to generate code that includes custom pragmas, use custom storage classes and memory sections. See “Control Data and Function Placement in Memory by Inserting Pragmas”.

See Also

`Simulink.LookupTable` | `Simulink.Breakpoint` | `Simulink.Parameter`

Related Examples

- “Exchange and Reuse Parameter Data Between Generated Code and Existing Code”
- “Reuse Parameter Data in Different Data Type Contexts” on page 14-90
- “Block Parameter Representation in the Generated Code” on page 14-43

Specify Instance-Specific Parameter Values for Reusable Referenced Model

When you use model referencing to break a large system into components, each component is a separate model. You can reuse a component by referring to it with multiple Model blocks. Each Model block is an instance of the component. You can then configure a block parameter (such as the **Gain** parameter of a Gain block) to use either the same value or a different value for each instance of the component. To use different values, create and use a model argument to set the value of the block parameter.

When you generate code from a model hierarchy that uses model arguments, the arguments appear in the code as formal parameters of the referenced model entry-point functions, such as the output (**step**) function. The generated code then passes the instance-specific parameter values, which you specify in each Model block, to the corresponding function calls.

Whether you use or don't use model arguments, you can use storage classes to configure block parameters to appear in the generated code as tunable global variables. You can also use storage classes to generate tunable model argument values, which the generated code stores in memory and passes to the function calls. You can then change the values during execution.

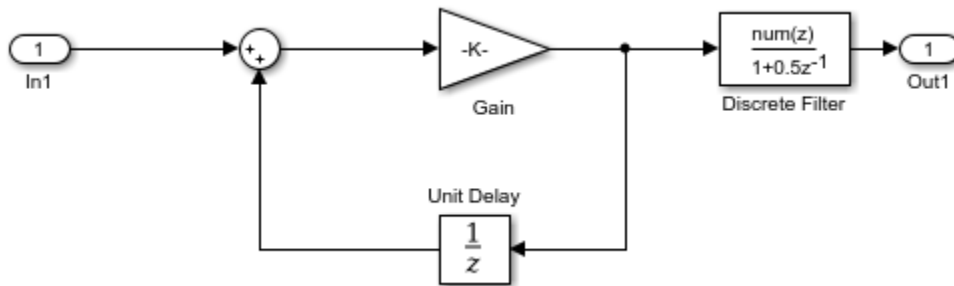
Pass Parameter Data to Referenced Model Entry-Point Functions as Arguments

Configure a referenced model to accept parameter data through formal parameters of the generated model entry-point function. This technique enables you to specify a different parameter value for each instance (Model block) of the referenced model.

Configure Referenced Model to Use Model Arguments

Create the model `ex_arg_code_ref`. This model represents a reusable algorithm.

```
open_system('ex_arg_code_ref')
```



In the Inport block dialog box, on the **Signal Attributes** tab, set **Data type** to **single**. Due to data type inheritance, the other signals in the model use the same data type.

```
set_param('ex_arg_code_ref/In1','OutDataTypeStr','single')
```

In the Model Explorer **Model Hierarchy** pane, expand the node **ex_arg_code_ref** and select **Model Workspace**.

Select **Add > Simulink Parameter** twice. Two Simulink.Parameter objects appear in the **Contents** pane.

Rename the objects as **gainArg** and **coeffArg**.

Set the **Value** property of the objects. For example, set them to **3.17** and **1.05**, respectively.

```
modelWorkspace = get_param('ex_arg_code_ref','ModelWorkspace');
assignin(modelWorkspace,'gainArg',Simulink.Parameter(3.17));
assignin(modelWorkspace,'coeffArg',Simulink.Parameter(1.05));
```

In the **Model Hierarchy** pane, select **Model Workspace**.

In the Dialog pane (the right-hand pane), set **Model arguments** to **coeffArg,gainArg**.

```
set_param('ex_arg_code_ref','ParameterArgumentNames','coeffArg,gainArg')
```

The screenshot shows the Simulink Model Workspace interface. On the left, a table lists data objects:

Name	Value	Dimensions	Description
gainArg	3.17	[1 1]	
coeffArg	1.05	[1 1]	

On the right, the 'Model Workspace' panel shows 'Workspace data' with 'Data source' set to 'Model File'. Below it, 'Model arguments (for referencing this model):' are listed as 'coeffArg,gainArg'.

In the `ex_arg_code_ref` model, in the Gain block dialog box, set **Gain** to `gainArg`.

```
set_param('ex_arg_code_ref/Gain', 'Gain', 'gainArg')
```

In the Discrete Filter block dialog box, set **Numerator** to `coeffArg`.

```
set_param('ex_arg_code_ref/Discrete Filter', 'Numerator', 'coeffArg')
```

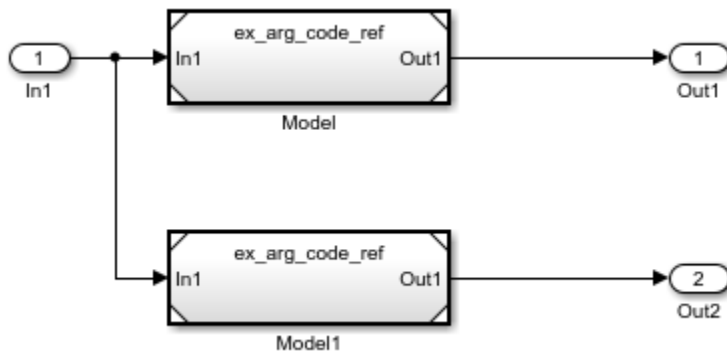
Save the `ex_arg_code_ref` model.

```
save_system('ex_arg_code_ref')
```

Specify Instance-Specific Parameter Values in Model Blocks

Create the model `ex_arg_code_ref`. This model uses multiple instances (Model blocks) of the reusable algorithm.

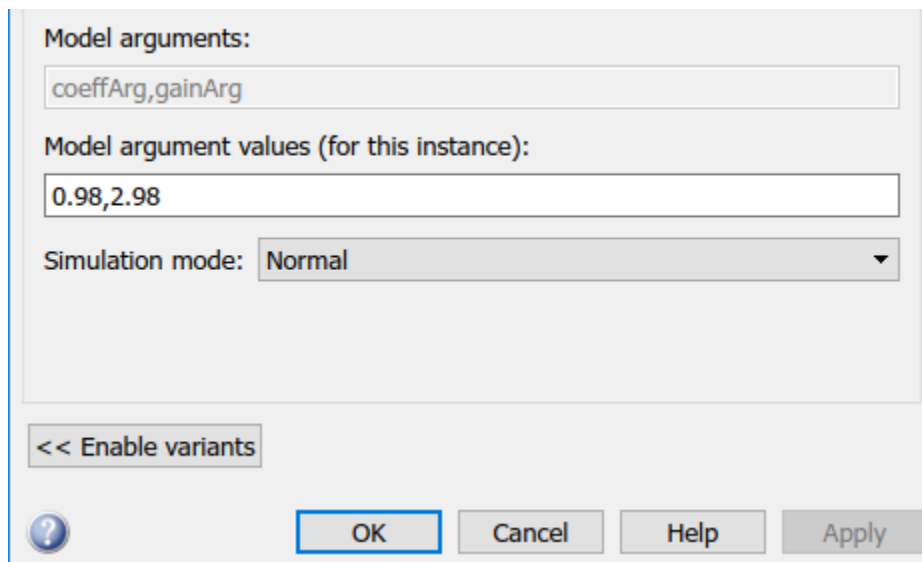
```
open_system('ex_arg_code')
```



Open the upper Model block dialog box.

Set **Model argument values** to 0.98,2.98.

```
set_param('ex_arg_code/Model1', 'ParameterArgumentValues', ...
    '0.98,2.98')
```



In the other Model block dialog box, set **Model argument values** to 1.11,3.34.

```
set_param('ex_arg_code/Model11', 'ParameterArgumentValues', ...
    '1.11,3.34')
```

Generate code from the top model.

```
rtwbuild('ex_arg_code')
```

```
### Successfully updated the model reference RTW target for model: ex_arg_code_ref
### Starting build procedure for model: ex_arg_code
### Successful completion of build procedure for model: ex_arg_code
```

The file `ex_arg_code_ref.c` defines the referenced model entry-point function, `ex_arg_code_ref`. The function has two formal parameters, `rtp_coeffArg` and `rtp_gainArg`, that correspond to the model arguments, `coeffArg` and `gainArg`. The

formal parameters use the data type `real32_T`, which corresponds to the data type `single` in Simulink.

```
file = fullfile('slprj','grt','ex_arg_code_ref','ex_arg_code_ref.c');
rtwdemodbtype(file,'/* Output and update for referenced model:',...
    'real32_T rtp_gainArg)',1,1)

/* Output and update for referenced model: 'ex_arg_code_ref' */
void ex_arg_code_ref(const real32_T *rtu_In1, real32_T *rty_Out1,
    DW_ex_arg_code_ref_f_T *localDW, real32_T rtp_coeffArg,
    real32_T rtp_gainArg)
```

The file `ex_arg_code.c` contains the definition of the top model entry-point function, `ex_arg_code`. This function calls the referenced model entry-point function, `ex_arg_code_ref`, and uses the model argument values that you specified (such as 1.11 and 3.34) as the values of `rtp_coeffArg` and `rtp_gainArg`.

```
file = fullfile('ex_arg_code_grt_rtw','ex_arg_code.c');
rtwdemodbtype(file,'/* ModelReference: '<Root>/Model' */',...
    '1.11F, 3.34F',1,1)

/* ModelReference: '<Root>/Model' */
ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out1,
    &(ex_arg_code_DW.Model_DWORK1.rtdw), 0.98F, 2.98F);

/* ModelReference: '<Root>/Model1' */
ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out2,
    &(ex_arg_code_DW.Model1_DWORK1.rtdw), 1.11F, 3.34F);
```

The formal parameters use the data type `real32_T` (`single`) because:

- 1 The block parameters in `ex_arg_code_ref` determine their data types through internal rules. For example, in the Gain block dialog box, on the **Parameter Attributes** tab, **Parameter data type** is set to **Inherit: Inherit via internal rule** (the default). In this case, the internal rule chooses the same data type as the input and output signals, `single`.
- 2 The model arguments in the model workspace use context-sensitive data typing because the value of the `Data Type` property is set to `auto` (the default). With this setting, the model arguments use the same data type as the block parameters, `single`.
- 3 The formal parameters in the generated code use the same data type as the model arguments, `single`.

Generate Tunable Argument Values

You can configure the instance-specific values in the Model blocks to appear in the generated code as tunable global variables. This technique enables you to store the parameter values for each instance in memory and to tune the values during code execution.

View the contents of the `ex_arg_code_ref` model workspace in Model Explorer.

Copy `gainArg` and `coeffArg` from the `ex_arg_code_ref` model workspace to the base workspace.

Rename `gainArg` as `gainForInst1`. Rename `coeffArg` as `coeffForInst1`.

```
gainForInst1 = getVariable(modelWorkspace, 'gainArg');  
gainForInst1 = copy(gainForInst1);  
coeffForInst1 = getVariable(modelWorkspace, 'coeffArg');  
coeffForInst1 = copy(coeffForInst1);
```

Copy `gainForInst1` and `coeffForInst1` as `gainForInst2` and `coeffForInst2`.

```
gainForInst2 = copy(gainForInst1);  
coeffForInst2 = copy(coeffForInst1);
```

Set the instance-specific parameter values by using the `Value` property of the parameter objects in the base workspace.

```
gainForInst1.Value = 2.98;  
coeffForInst1.Value = 0.98;
```

```
gainForInst2.Value = 3.34;  
coeffForInst2.Value = 1.11;
```

For the new parameter objects, set `StorageClass` to `ExportedGlobal`. This setting causes the parameter objects to appear in the generated code as tunable global variables.

```
gainForInst1.StorageClass = 'ExportedGlobal';  
coeffForInst1.StorageClass = 'ExportedGlobal';  
gainForInst2.StorageClass = 'ExportedGlobal';  
coeffForInst2.StorageClass = 'ExportedGlobal';
```

In the top model, `ex_arg_code`, open the upper Model block dialog box.

Set **Model argument values** to `coeffForInst1, gainForInst1`.

```
set_param('ex_arg_code/Model1','ParameterArgumentValues',...
    'coeffForInst1,gainForInst1')
```

The screenshot shows a dialog box with the following fields and controls:

- Model arguments:** A text input field containing "coeffArg,gainArg".
- Model argument values (for this instance):** A text input field containing "coeffForInst1,gainForInst1".
- Simulation mode:** A dropdown menu currently showing "Normal".
- Buttons:** A "<< Enable variants" button on the left, and "OK", "Cancel", "Help", and "Apply" buttons on the right.
- Help:** A question mark icon in a circle at the bottom left.

In the other Model block dialog box, set **Model argument values** to `coeffForInst2,gainForInst2`.

```
set_param('ex_arg_code/Model1','ParameterArgumentValues',...
    'coeffForInst2,gainForInst2')
```

Generate code from the top model.

```
rtwbuild('ex_arg_code')

### Starting build procedure for model: ex_arg_code
### Successful completion of build procedure for model: ex_arg_code
```

The file `ex_arg_code.c` defines the global variables that correspond to the parameter objects in the base workspace.

```
file = fullfile('ex_arg_code_grt_rtw','ex_arg_code.c');
rtwdmodbtype(file,'/* Exported block parameters */',...
    '/* Block states (auto storage) */',1,0)

/* Exported block parameters */
```

```

real32_T coeffForInst1 = 0.98F;      /* Variable: coeffForInst1
                                     * Referenced by: '<Root>/Model1'
                                     */
real32_T coeffForInst2 = 1.11F;     /* Variable: coeffForInst2
                                     * Referenced by: '<Root>/Model1'
                                     */
real32_T gainForInst1 = 2.98F;      /* Variable: gainForInst1
                                     * Referenced by: '<Root>/Model1'
                                     */
real32_T gainForInst2 = 3.34F;     /* Variable: gainForInst2
                                     * Referenced by: '<Root>/Model1'
                                     */

```

In each call to `ex_arg_code_ref`, the top model algorithm uses the global variables to set the values of the formal parameters.

```

rtwdemodbtype(file, /* ModelReference: '<Root>/Model1' */',...
              'gainForInst2);',1,1)

/* ModelReference: '<Root>/Model1' */
ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out1,
                &(ex_arg_code_DW.Model1_DWORK1.rtdw), coeffForInst1,
                gainForInst1);

/* ModelReference: '<Root>/Model1' */
ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out2,
                &(ex_arg_code_DW.Model1_DWORK1.rtdw), coeffForInst2,
                gainForInst2);

```

The global variables in the generated code use the data type `real32_T` (single) because:

- 1 The parameter objects in the base workspace use context-sensitive data typing because the `DataType` property is set to `auto` (the default). With this setting, the parameter objects in the base workspace use the same data type as the model arguments, `single`.
- 2 The global variables in the generated code use the same data type as the parameter objects in the base workspace.

Group Multiple Model Arguments into Single Structure

Use the Model Explorer to copy `gainArg` and `coeffArg` from the `ex_arg_code_ref` model workspace into the base workspace.

```
temp = getVariable(modelWorkspace, 'gainArg');
gainArg = copy(temp);
temp = getVariable(modelWorkspace, 'coeffArg');
coeffArg = copy(temp);
```

At the command prompt, combine these two parameter objects into a structure, `structArg`.

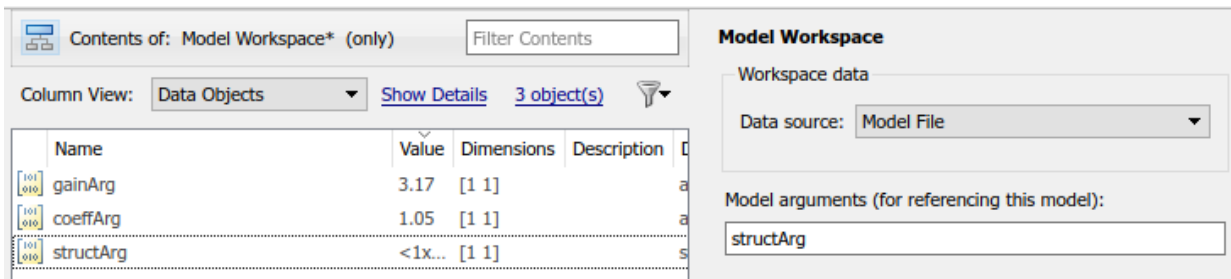
```
structArg = Simulink.Parameter(struct('gain', gainArg.Value, ...
    'coeff', coeffArg.Value));
```

Use the Model Explorer to move `structArg` into the model workspace.

```
assignin(modelWorkspace, 'structArg', copy(structArg));
clear structArg gainArg coeffArg
```

In the **Model Hierarchy** pane, select the `ex_arg_code_ref` model workspace. In the Dialog pane, set **Model arguments** to `structArg`.

```
set_param('ex_arg_code_ref', 'ParameterArgumentNames', 'structArg')
```



The screenshot shows the MATLAB Model Explorer interface. The left pane, titled 'Contents of: Model Workspace* (only)', displays a table of objects in the workspace. The table has columns for Name, Value, Dimensions, and Description. The objects listed are gainArg, coeffArg, and structArg. The right pane, titled 'Model Workspace', shows the configuration for the workspace. The 'Workspace data' section has 'Data source' set to 'Model File'. The 'Model arguments (for referencing this model):' section contains the text 'structArg'.

Name	Value	Dimensions	Description
gainArg	3.17	[1 1]	
coeffArg	1.05	[1 1]	
structArg	<1x...	[1 1]	

In the `ex_arg_code_ref` model, set the **Gain** parameter of the Gain block to `structArg.gain`.

```
set_param('ex_arg_code_ref/Gain', 'Gain', 'structArg.gain')
```

In the Discrete Filter block dialog box, set **Numerator** to `structArg.coeff`.

```
set_param('ex_arg_code_ref/Discrete Filter', ...
    'Numerator', 'structArg.coeff')
```

At the command prompt, combine the four parameter objects in the base workspace into two structures. Each structure stores the parameter values for one instance of `ex_arg_code_ref`.

```
structForInst1 = Simulink.Parameter(struct('gain',gainForInst1.Value,...
    'coeff',coeffForInst1.Value));
```

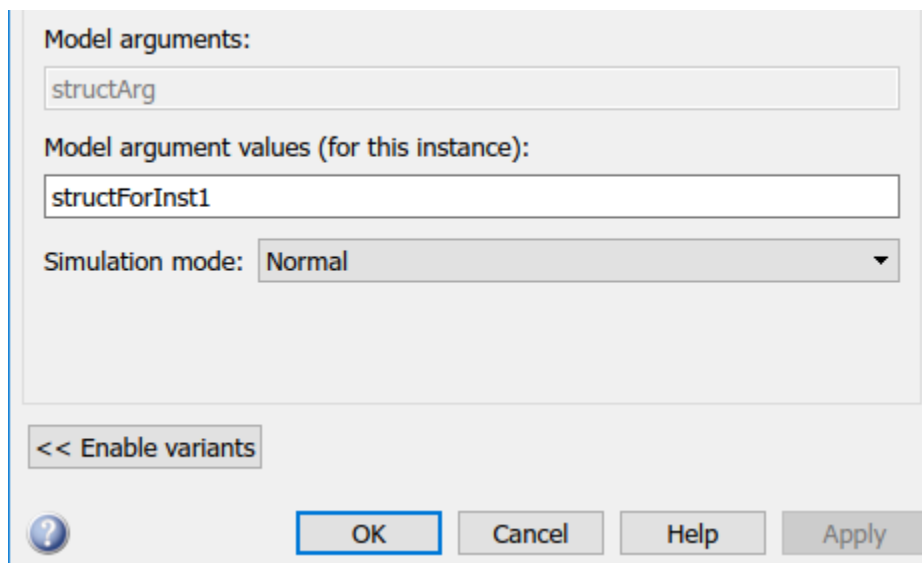
```
structForInst2 = Simulink.Parameter(struct('gain',gainForInst2.Value,...
    'coeff',coeffForInst2.Value));
```

Apply the storage class `ExportedGlobal` to the parameter objects.

```
structForInst1.StorageClass = 'ExportedGlobal';
structForInst2.StorageClass = 'ExportedGlobal';
```

In the top model, in the upper Model block dialog box, set **Model argument values** to `structForInst1`.

```
set_param('ex_arg_code/Model1','ParameterArgumentValues','structForInst1')
```



In the other Model block dialog box, set **Model argument values** to `structForInst2`.

```
set_param('ex_arg_code/Model1','ParameterArgumentValues','structForInst2')
```

Use the function `Simulink.Bus.createObject` to create a `Simulink.Bus` object. The hierarchy of elements in the object matches the hierarchy of the structure fields. The default name of the object is `s1Bus1`.

```
Simulink.Bus.createObject(structForInst1.Value);
```

Rename the bus object as `myParamStructType` by copying it.

```
myParamStructType = copy(s1Bus1);
```

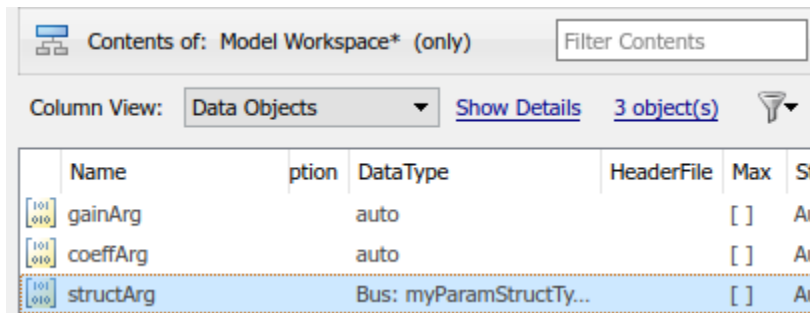
Set the data type of the parameter objects in the base workspace by using the bus object.

```
structForInst1.DataType = 'Bus: myParamStructType';
structForInst2.DataType = 'Bus: myParamStructType';
```

Use the Model Explorer to view the contents of the `ex_arg_code_ref` model workspace.

Set **Column View** to **Data Objects**. For the parameter object `structArg`, set **DataType** to **Bus: myParamStructType**.

```
temp = getVariable(modelWorkspace, 'structArg');
temp = copy(temp);
temp.DataType = 'Bus: myParamStructType';
assignin(modelWorkspace, 'structArg', copy(temp));
```



Save the `ex_arg_code_ref` model.

```
save_system('ex_arg_code_ref')
```

When you use structures to group parameter values, you cannot take advantage of context-sensitive data typing to control the data types of the fields of the structures (for example, the fields of `structForInst1`). However, you can use the properties of the bus object to control the field data types.

Set the data type of the elements in the bus object to `single`. The corresponding fields in the structures (such as `structForInst1` and `structArg`) use the same data type.

```
myParamStructType.Elements(1).DataType = 'single';
myParamStructType.Elements(2).DataType = 'single';
```

Generate code from the top model, `ex_arg_code`.

```
rtwbuild('ex_arg_code')

### Successfully updated the model reference RTW target for model: ex_arg_code_ref
### Starting build procedure for model: ex_arg_code
### Successful completion of build procedure for model: ex_arg_code
```

The file `ex_arg_code_types.h` defines the structure type `myParamStructType`, which corresponds to the `Simulink.Bus` object.

```
file = fullfile('ex_arg_code_grt_rtw', 'ex_arg_code_types.h');
rtwdemodbtype(file, 'typedef struct {' , '}' myParamStructType; ',1,1)
```

```
typedef struct {
    real32_T gain;
    real32_T coeff;
} myParamStructType;
```

In the file `ex_arg_code_ref.c`, the referenced model entry-point function has a formal parameter, `rtp_structArg`, that correspond to the model argument `structArg`.

```
file = fullfile('slprj', 'grt', 'ex_arg_code_ref', 'ex_arg_code_ref.c');
rtwdemodbtype(file, '/* Output and update for referenced model:', ...
    '*rtp_structArg)',1,1)
```

```
/* Output and update for referenced model: 'ex_arg_code_ref' */
void ex_arg_code_ref(const real32_T *rtu_In1, real32_T *rty_Out1,
    DW_ex_arg_code_ref_f_T *localDW, const myParamStructType
    *rtp_structArg)
```

The file `ex_arg_code.c` defines the global structure variables that correspond to the parameter objects in the base workspace.

```
file = fullfile('ex_arg_code_grt_rtw', 'ex_arg_code.c');
rtwdemodbtype(file, '/* Exported block parameters */', ...
    '/* Block states (auto storage) */',1,0)
```

```
/* Exported block parameters */
myParamStructType structForInst1 = {
    2.98F,
    0.98F
```



```

} ;                               /* Variable: structForInst1
                                   * Referenced by: '<Root>/Model'
                                   */

myParamStructType structForInst2 = {
    3.34F,
    1.11F
} ;                               /* Variable: structForInst2
                                   * Referenced by: '<Root>/Model1'
                                   */

```

The top model algorithm in the file `ex_arg_code.c` passes the addresses of the structure variables to the referenced model entry-point function.

```

file = fullfile('ex_arg_code_grt_rtw','ex_arg_code.c');
rtwdemodbtype(file, /* ModelReference: '<Root>/Model' */,...
    '&structForInst2'); ,1,1)

/* ModelReference: '<Root>/Model' */
ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out1,
    &(ex_arg_code_DW.Model_DWORK1.rtdw), &structForInst1);

/* ModelReference: '<Root>/Model1' */
ex_arg_code_ref(&ex_arg_code_U.In1, &ex_arg_code_Y.Out2,
    &(ex_arg_code_DW.Model1_DWORK1.rtdw), &structForInst2);

```

Control Data Types of Model Arguments and Argument Values

When you use model arguments, you can apply a data type to:

- The block parameters that use the arguments (for certain blocks, such as those in the Discrete library).
- The arguments in the referenced model workspace.
- The argument values that you specify in Model blocks.

To generate efficient code by eliminating unnecessary typecasts and C shifts, consider using inherited and context-sensitive data typing to match the data types.

- In the model workspace, use a MATLAB variable whose data type is `double` or a parameter object whose `DataType` property is set to `auto`. In this case, the variable or object uses the same data type as the block parameter.

- When you set the argument values in Model blocks, take advantage of context-sensitive data typing. To set an argument value, use an untyped value.
 - A literal number such as `15.23`. Do not use a typed expression such as `single(15.23)`.
 - A MATLAB variable whose data type is `double`.
 - A `Simulink.Parameter` object whose `DataType` property is set to `auto`.

In these cases, the number, variable, or object uses the same data type as the model argument in the referenced model workspace. If you also configure the model argument to use context-sensitive data typing, you can control the data types of the block parameter, the argument, and the argument value by specifying the type only for the block parameter.

For basic information about controlling parameter data types, see “Parameter Data Types in the Generated Code” on page 14-76.

Use Model Argument in Different Data Type Contexts

If you use a model argument to set multiple block parameter values, and the data types of the block parameters differ, you cannot use context-sensitive data typing (`double` or `auto`) for the argument in the model workspace. You must explicitly specify a data type for the argument. For example, if the argument in the model workspace is a parameter object (such as `Simulink.Parameter`), set the `DataType` property to a value other than `auto`. For more information about this situation, see “Reuse Parameter Data in Different Data Type Contexts” on page 14-90.

In this case, you can continue to take advantage of context-sensitive data typing to control the data type of the argument values that you specify in Model blocks. Each argument value uses the data type that you specify for the corresponding argument in the model workspace.

Related Examples

- “Code Generation of Referenced Models” on page 8-2
- “Block Parameter Representation in the Generated Code” on page 14-43
- “Parameterize Instances of a Reusable Referenced Model”
- “Organize Block Parameter Values into Structures in the Generated Code” on page 14-94

- “Parameter Data Types in the Generated Code” on page 14-76

Parameter Data Types in the Generated Code

The data type of a block parameter (such as the **Gain** parameter of a Gain block), numeric MATLAB variable, or `Simulink.Parameter` object determines the data type that the corresponding entity in the generated code uses (for example, a global variable or an argument of a function). To generate more efficient code, you can match parameter data types with signal data types or store parameters in smaller data types.

For basic information about setting block parameter data types in a model, see “Control Block Parameter Data Types”.

Significance of Parameter Data Types

The data type that a block parameter, MATLAB variable, or parameter object uses determines the data type that the generated code uses to store the parameter value in memory. For example:

- If you set the model configuration parameter **Default parameter behavior** (see “Default parameter behavior”) to **Tunable**, the **Gain** parameter of a Gain block appears in the generated code as a field of a global structure that stores parameter data. If you apply the data type `single` to the block parameter in the model, the structure field uses the corresponding data type (`real32_T`).
- If you apply the storage class `ExportedGlobal` to a `Simulink.Parameter` object, the object appears in the generated code as a separate global variable. If you set the `DataType` property of the object to `int8`, the global variable in the code uses the corresponding data type (`int8_T`).
- If you configure a `Simulink.Parameter` object in a model workspace as a model argument, the object appears in the generated code as a formal parameter of a model entry-point function, such as the `step` function. The `DataType` property of the object determines the data type of the formal parameter.

Other than determining the data type that the generated code uses to store parameter values in memory, the data type of a parameter, variable, or object can also:

- Cause the block to cast the value of the parameter prior to code generation. The cast can result in overflow, underflow, or quantization.
- Cause the generated code to include extra code, for example saturation code.

Parameter Data Type Mismatch

When the data types of block parameters, workspace variables, and signals differ, blocks can use typecasts to reconcile the data type mismatches. These typecasts can cause the generated code algorithm, including the model `step` function, to include explicit casts to reconcile mismatches in storage data type and C bit shifts to reconcile mismatches in fixed-point scaling.

Parameter data type mismatches can occur when:

- The data type that you specify for a MATLAB variable or parameter object (`Simulink.Parameter`) differs from the data type of a block parameter. The block parameter typecasts the value of the variable or object.
- The data type that you specify for an initial value differs from the data type of the initialized signal or state.
- The data type that you specify for a block parameter differs from the data type of the signal or signals that the parameter operates on. Some blocks typecast the parameter to perform the operation. For example, the Gain block performs this typecast.

If you configure a variable or object to use bias or fractional fixed-point slope, the block parameter cannot perform the typecast. In this case, you must match the data type of the variable or parameter object with the data type of the block parameter. Use one of these techniques:

- Use context-sensitive data typing for the variable or parameter object. For a MATLAB variable, use a `double` number to set the value of the variable. For a parameter object, set the `DataType` property to `auto`.
- Use a `Simulink.AliasType` or `Simulink.NumericType` object to set the data type of the block parameter and the data type of a parameter object.

Use this technique when you cannot rely on context-sensitive data typing, for example, when you use the field of a structure to set the value of the block parameter.

- Manually specify the same data type for the block parameter and the variable or parameter object.

Use this technique to reduce the dependence of the model on inherited and context-sensitive data types and on external variables and objects.

For blocks that access parameter data through pointer or reference in the generated code, if you specify a different data type of the workspace variable and block parameter, the

generated code implicitly casts the data type of the variable to the data type of the block parameter. Note, that an implicit cast requires a data copy which could significantly increase RAM consumption and slow down code execution speed for large data sets. For example, Lookup Table blocks often access large vectors or matrices through pointer or reference in the generated code.

For information about matching parameter data types when you use model arguments, see “Control Data Types of Model Arguments and Argument Values” on page 14-73.

Detect Downcast and Loss of Precision due to Data Type Mismatches

You can configure diagnostic configuration parameters to detect unintentional data type mismatches that result in quantization and loss of parameter precision. See “Model Configuration Parameters: Data Validity Diagnostics”.

Considerations for Other Modeling Patterns

When you use specific modeling patterns and constructs such as fixed-point data types, parameter structures, and lookup table objects, use different techniques to control parameter data types.

- “Tunable Parameters and Best-Precision Fixed-Point Scaling” on page 14-78
- “Control Data Types of Structure Fields” on page 14-79
- “Control Data Types of Lookup Table Objects” on page 14-79

Tunable Parameters and Best-Precision Fixed-Point Scaling

To apply best-precision fixed-point scaling to a tunable block parameter or parameter object, you can use the Fixed-Point Tool to autoscale an entire system or use the Data Type Assistant to configure individual parameters or objects. See “Calculate Best-Precision Fixed-Point Scaling for Tunable Block Parameters”.

If a tunable parameter uses best-precision fixed-point scaling, Simulink chooses a data type based on the minimum and maximum values that you specify for the parameter. You can specify these values in the block dialog box that uses the parameter or in the properties of a Simulink.Parameter object.

If you do not specify a minimum or maximum, Simulink chooses a data type based on the value of the parameter. The chosen scaling might restrict the range of possible tuning values. Therefore, it is a best practice to specify minimum and maximum values for each tunable parameter.

A tunable parameter can use best-precision scaling even if you do not specify it in the parameter data type. For example, the **Gain** block can choose a best-precision scaling if the **Parameter data type** in the block dialog box is set to **Inherit: Inherit via internal rule**. This setting is the default for the block.

Control Data Types of Structure Fields

When you use a structure as the value of a block parameter (for example to initialize a bus signal), or when you organize multiple block parameter values into a single structure, you can create a `Simulink.Bus` object to use as the data type of a `Simulink.Parameter` object. You can then control the data types of individual fields in the structure. See “Control Field Data Types and Characteristics by Creating Parameter Object” and “Control Data Types of Initial Condition Structure Fields”.

Control Data Types of Lookup Table Objects

When you use `Simulink.LookupTable` and `Simulink.Breakpoint` objects to store table and breakpoint data for a lookup table block, to control the data types of the table and breakpoint data, use one of these techniques:

- Set the **Value** property of the embedded `Simulink.lookuptable.Table` and `Simulink.lookuptable.Breakpoint` objects by using untyped expressions such as `[1 2 3]`, which returns a `double` vector. To control the data type, set the **DataType** property to a value other than `auto`.

Use this technique to separate the value of the table or breakpoint data from the data type, which can improve readability and understanding of your design. You can then use a `Simulink.NumericType` or `Simulink.AliasType` object to:

- Customize the name of the data type in the generated code.
- Match the data type of the table or breakpoint data with the data type of a signal in the model.
- Set the **Value** property of the embedded objects by using typed expressions such as `single([1 2 3])`. To use a fixed-point data type, set the **Value** property with an `fi` object.

Set the **DataType** property of the embedded objects to the default value, `auto`. The table and breakpoint data then acquire the data type that you use to set the **Value** property.

Use this technique to store the data type information in the `Value` property, which can simplify the way you interact with the `Simulink.LookupTable` and `Simulink.Breakpoint` objects. You can leave the `DataType` property at the default value.

When you later change the breakpoint or table data in the `Value` property, preserve the data type information by using a typed expression. Alternatively, if you use a command at the command prompt or a script to change the data, to avoid using a typed expression, use subscripted assignment, `(:)`.

```
myLUTObject.Table.Value(:) = [4 5 6];
```

When you change the data stored in the `Value` property, if you do not use a typed expression or subscripted assignment, you lose the data type information.

When blocks in a subsystem use `Simulink.LookupTable` or `Simulink.Breakpoint` objects, you cannot set data type override (see “Control Fixed-Point Instrumentation and Data Type Override”) only on the subsystem. Instead, set data type override on the entire model.

Related Examples

- “Generate Efficient Code by Specifying Data Types for Block Parameters” on page 14-81
- “Reuse Parameter Data in Different Data Type Contexts” on page 14-90
- “Data Types Supported by Simulink”
- “Control Signal Data Types”
- “Block Parameter Representation in the Generated Code” on page 14-43

Generate Efficient Code by Specifying Data Types for Block Parameters

To generate more efficient code, match the data types of block parameters (such as the **Gain** parameter of a Gain block) with signal data types. Alternatively, you can store parameters in smaller data types.

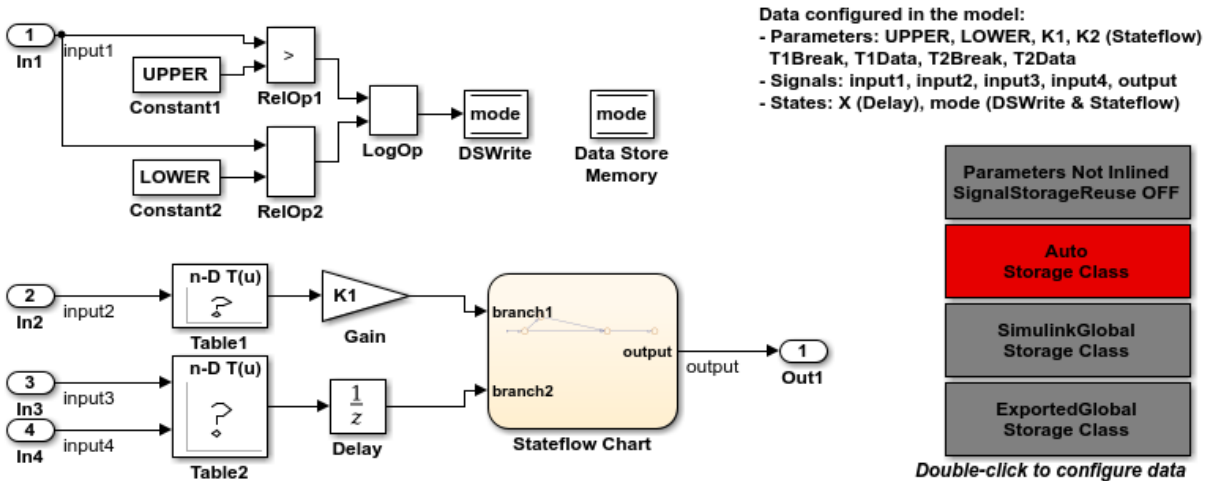
Eliminate Unnecessary Typecasts and Shifts by Matching Data Types

These examples show how to generate efficient code by configuring a block parameter to use the same data type as a signal that the block operates on.

Store Data Type Information in Model

Open the example model `rtwdemo_basicsc`.

`rtwdemo_basicsc`



Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

Advanced data packaging using Simulink data objects ... (double-click)

Open the **Gain** block dialog box. The input signal of this block uses the data type `single`.

View the **Parameter Attributes** tab. **Parameter data type** is set to **Inherit: Same as input**. The **Gain** parameter of this block uses the same data type as the input signal.

At the command prompt, convert the numeric variable `K1` to a `Simulink.Parameter` object.

```
K1 = Simulink.Parameter(2);
```

Configure `K1` to appear in the generated code as a global variable by applying the storage class `ExportedGlobal`.

```
K1.CoderInfo.StorageClass = 'ExportedGlobal';
```

On the **Main** tab, right-click the value of the **Gain** parameter, `K1`. Select **Open Variable**. The value of the **Data type** property is `auto`, which means the parameter object acquires its data type from the block parameters that use the object.

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')

### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

The generated file `rtwdemo_basicsc.c` defines the global variable `K1` by using the data type `real32_T`, which corresponds to the data type `single` in Simulink.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.c');
rtwdemodbtype(file,'/* Exported block parameters */','real32_T K1 = 2.0F;',1,1)

/* Exported block parameters */
real32_T K1 = 2.0F;                                /* Variable: K1
```

The generated code algorithm in the model `step` function uses `K1` directly without typecasting.

```
rtwdemodbtype(file,' rtwdemo_basicsc_DW.X = K1',...
```

```
' rtCP_Table1_bp01Data, rtCP_Table1_tableData',1,1)

rtwdemo_basicsc_DW.X = K1 * look1_iflf_binlx(rtwdemo_basicsc_U.input2,
    rtCP_Table1_bp01Data, rtCP_Table1_tableData, 10U);
```

In the **Gain** block dialog box, on the **Parameter Attributes** tab, you can optionally set **Parameter data type** to **Inherit: Inherit via internal rule** (the default). In this case, the block parameter chooses the same data type as the input signal (**single**). However, when you use **Inherit: Inherit via internal rule**, under other circumstances (for example, when you use fixed-point data types) the block parameter might choose a different data type.

Store Data Type Information in Parameter Object

When you use a `Simulink.Parameter` object to export or import parameter data from the generated code to your custom code, for example by applying the storage class `ImportedExtern`, you can specify data type information in the parameter object. To match the data type of the parameter object with a signal data type, create a `Simulink.NumericType` or `Simulink.AliasType` object. You can strictly control the data type that the parameter object uses in the generated code, eliminating the risk that Simulink chooses a different data type when you make changes to the model.

At the command prompt, create a `Simulink.NumericType` object that represents the data type `single`.

```
myType = Simulink.NumericType;
myType.DataTypeMode = 'Single';
```

Use this data type object as the data type of the parameter object.

```
K1.DataType = 'myType';
```

Use the data type object to set the output data type of the Inport block named `In2`. Due to data type propagation, the input signal of the Gain block also uses this data type.

```
set_param('rtwdemo_basicsc/In2','OutDataTypeStr','myType')
```

In the Gain block dialog box, on the **Parameter Attributes** tab, set **Parameter data type** to **Inherit: Inherit from 'Gain'**.

```
set_param('rtwdemo_basicsc/Gain','ParamDataTypeStr','Inherit: Inherit from ''Gain''')
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')
```

```
### Starting build procedure for model: rtwdemo_basicsc  
### Successful completion of build procedure for model: rtwdemo_basicsc
```

The global variable K1 continues to use the data type `real32_T`.

```
file = fullfile('rtwdemo_basicsc_grt_rtw', 'rtwdemo_basicsc.c');  
rtwdemodbtype(file, '/* Exported block parameters */', 'real32_T K1 = 2.0F;', 1, 1)
```

```
/* Exported block parameters */  
real32_T K1 = 2.0F;                               /* Variable: K1
```

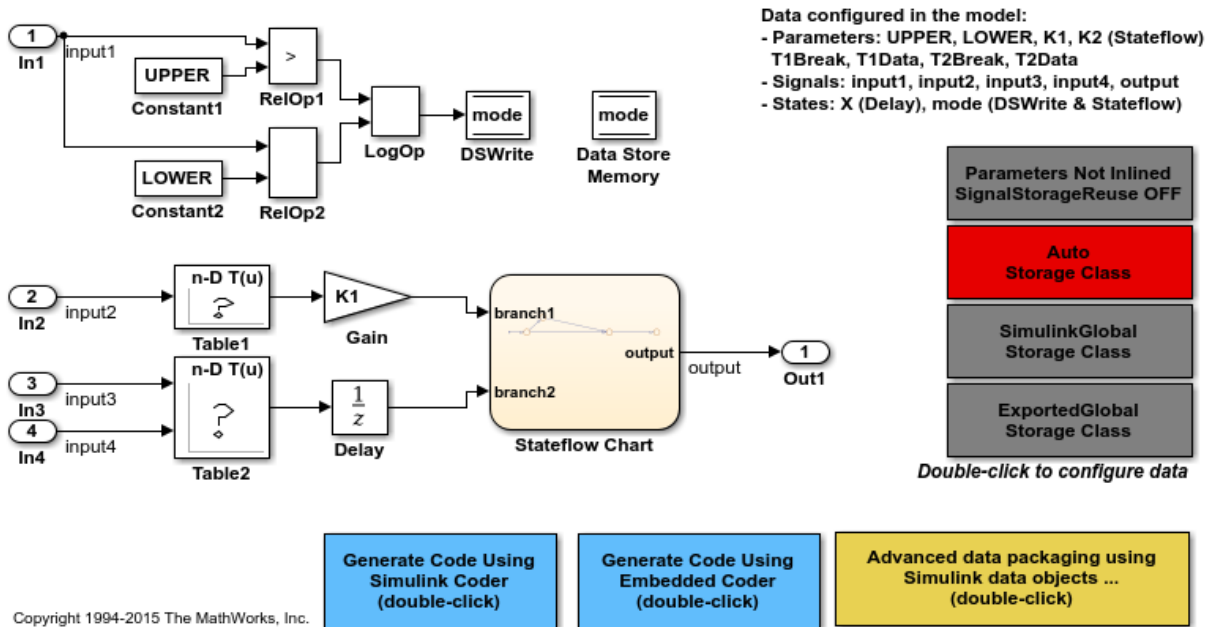
Reduce Memory Consumption by Storing Parameter Value in Small Data Type

When you use a parameter object (for example, `Simulink.Parameter`) to set block parameter values, you can configure the object to appear in the generated code as a tunable global variable. By default, the parameter object and the corresponding global variable typically uses the same data type as the signal or signals on which the block operates. For example, if the input signal of a Gain block uses the data type `int16`, the parameter object typically use the same data type. To reduce the amount of memory that this variable consumes, specify that the variable use a smaller integer data type such as `int8`.

Store Parameter Value in Integer Data Type

Open the example model `rtwdemo_basicsc`.

```
rtwdemo_basicsc
```



Copyright 1994-2015 The MathWorks, Inc.

In the model, select **Display > Signals and Ports > Port Data Types**. Many of the signals in the model use the data type `single`.

Open the Gain block dialog box. The block uses the MATLAB variable `K1`, which is in the base workspace, to set the value of the **Gain** parameter. The value of the variable is 2.

At the command prompt, convert `K1` into a `Simulink.Parameter` object. Configure the object to appear in the generated code as a global variable by applying the storage class `ExportedGlobal`.

```
K1 = Simulink.Parameter(2);
K1.CoderInfo.StorageClass = 'ExportedGlobal';
```

Configure the parameter object to use the data type `int8`.

```
K1.DataType = 'int8';
```

In the Gain block dialog box, on the **Parameter Attributes** tab, set **Parameter data type** to **Inherit: Inherit from 'Gain'**. With this setting, the **Gain** parameter of the block inherits the `int8` data type from the parameter object.

```
set_param('rtwdemo_basicsc/Gain','ParamDataTypeStr',...
    'Inherit: Inherit from ''Gain''')
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc')
```

```
### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

The generated file `rtwdemo_basicsc.c` defines the global variable `K1` by using the data type `int8_T`, which corresponds to the data type `int8` in Simulink.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.c');
rtwdemodbtype(file,'/* Exported block parameters */','int8_T K1 = 2;',1,1)
```

```
/* Exported block parameters */
int8_T K1 = 2;                                /* Variable: K1
```

The code algorithm in the model `step` function uses `K1` to calculate the output of the Gain block. The algorithm casts `K1` to the data type `real32_T` (single) because the signals involved in the calculation use the data type `real32_T`.

```
rtwdemodbtype(file,' rtwdemo_basicsc_DW.X = ','10U);',1,1)
```

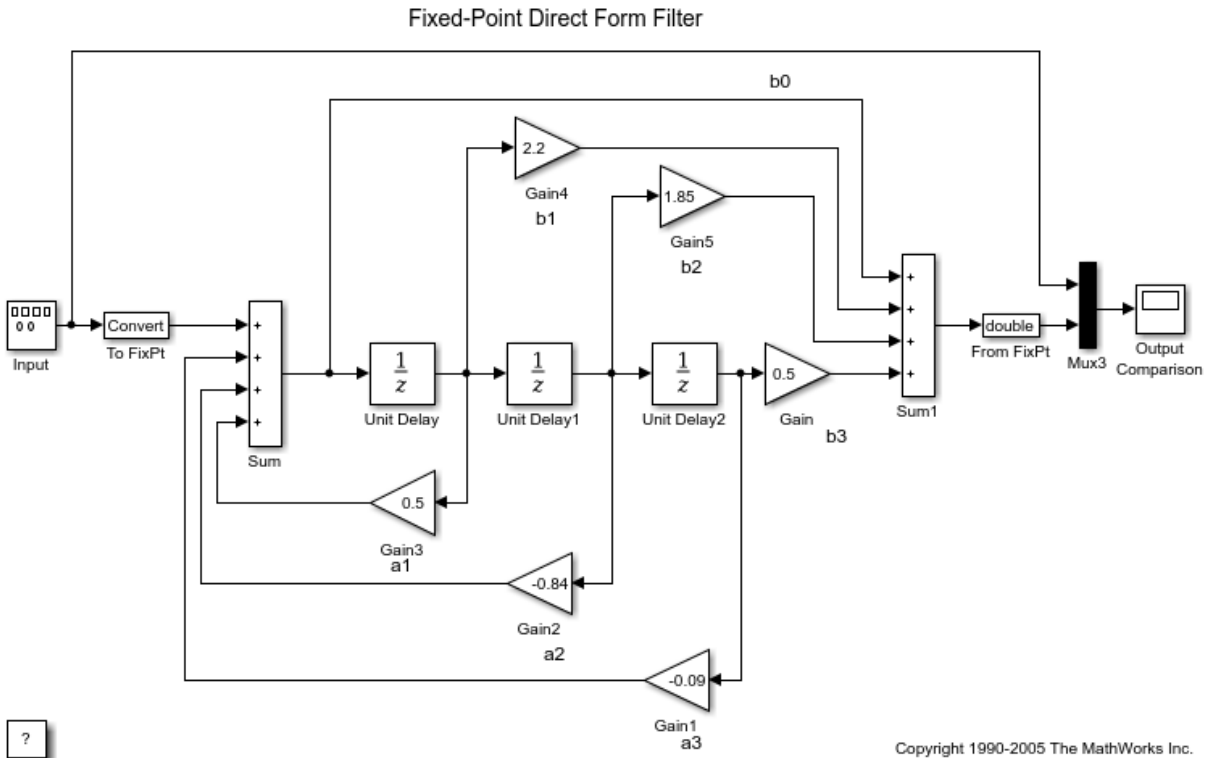
```
rtwdemo_basicsc_DW.X = (real32_T)K1 * look1_if1f_binlx
    (rtwdemo_basicsc_U.input2, rtCP_Table1_bp01Data, rtCP_Table1_tableData,
    10U);
```

Store Fixed-Point Parameter Value in Smaller Integer Data Type

Suppose you configure the signals in your model to use fixed-point data types. You want a gain parameter to appear in the generated code as a tunable global variable. You know the range of real-world values that you expect the parameter to assume (for example, between 0 and 4). If you can meet your application requirements despite reduced precision, you can reduce memory consumption by configuring the parameter to use a different data type than the input and output signals of the block.

Open the example model `fxpdemo_direct_form2`.

```
fxpdemo_direct_form2
```



Update the model diagram. Signals in this model use signed fixed-point data types with a word length of 16 and binary-point-only scaling.

Open the Gain5 block dialog box. The **Gain** parameter is set to 1.85. Suppose you want to configure this parameter.

Set **Gain** to `myGainParam` and click **Apply**.

Right-click `myGainParam` and select **Create Variable**.

In the Create New Data dialog box, set **Value** to `Simulink.Parameter(1.85)` and click **Create**. The `Simulink.Parameter` object `myGainParam` appears in the base workspace.

In the `myGainParam` property dialog box, set **Storage class** to `ExportedGlobal` and click **OK**. With this setting, `myGainParam` appears in the generated code as a global variable.

In the block dialog box, on the **Parameter Attributes** tab, set **Parameter minimum** to 0 and **Parameter maximum** to 4.

Set **Parameter data type** to `fixdt(0,8)` and click **Apply**.

Click the **Show Data Type Assistant** button. The Data Type Assistant shows that the expression `fixdt(0,8)` specifies an unsigned fixed-point data type with a word length of 8 and best-precision scaling. When you simulate or generate code, the block parameter chooses a fraction length (scaling) that enables the data type to represent values between the parameter minimum and maximum (0 and 4) with the best possible precision.

In the Data Type Assistant, set **Scaling** to `Binary point`. Click **Calculate Best-Precision Scaling**, **Fixed-point details**, and **Refresh Details**. The information under **Fixed-point details** shows that a fraction length of 5 can represent the parameter values with a precision of 0.03125.

Set **Scaling** back to `Best precision` and click **OK**. In this example, when you simulate or generate code, the block parameter chooses a fraction length of 5.

You can use these commands at the command prompt to create the object and configure the block:

```
myGainParam = Simulink.Parameter(1.85);
myGainParam.CoderInfo.StorageClass = 'ExportedGlobal';
set_param('fxpdemo_direct_form2/Gain5', 'Gain', 'myGainParam')
set_param('fxpdemo_direct_form2/Gain5', 'ParamMin', '0', 'ParamMax', '4')
set_param('fxpdemo_direct_form2/Gain5', 'ParamDataTypes', 'fixdt(0,8)')
```

Configure the model to produce a code generation report. To reduce clutter in the Command Window, clear the configuration parameter **Verbose build**.

```
set_param('fxpdemo_direct_form2', 'GenerateReport', 'on', ...
    'LaunchReport', 'on', 'RTWVerbose', 'off')
```

Generate code from the model.

```
rtwbuild('fxpdemo_direct_form2')

### Starting build procedure for model: fxpdemo_direct_form2
```


Code generation utilizes device specific information (e.g., microprocessor word sizes)

```
### Successful completion of build procedure for model: fxdemo_direct_form2
```

The generated file `fxpdemo_direct_form2.c` defines the global variable `myGainParam` by using the data type `uint8_T`, which corresponds to the specified word length, 8. The code initializes the variable by using an integer value that, given the fraction length of 5, represents the real-world parameter value 1.85.

```
file = fullfile('fxpdemo_direct_form2_grt_rtw','fxpdemo_direct_form2.c');
rtwdemodbtype(file,'/* Exported block parameters */','uint8_T myGainParam = 59U;',1,1)
```

```
/* Exported block parameters */
uint8_T myGainParam = 59U;          /* Variable: myGainParam
```

The code algorithm uses `myGainParam` to calculate the output of the Gain5 block. The algorithm uses a C shift to scale the result of the calculation.

```
rtwdemodbtype(file,'/* Gain: '<Root>/Gain5' */',...
'/* Gain: '<Root>/Gain' incorporates:',1,0)
```

```
/* Gain: '<Root>/Gain5' */
fxpdemo_direct_form2_B.Gain5 = (int16_T)(myGainParam *
fxpdemo_direct_form2_B.UnitDelay1 >> 5);
```

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Parameter Data Types in the Generated Code” on page 14-76
- “Reuse Parameter Data in Different Data Type Contexts” on page 14-90

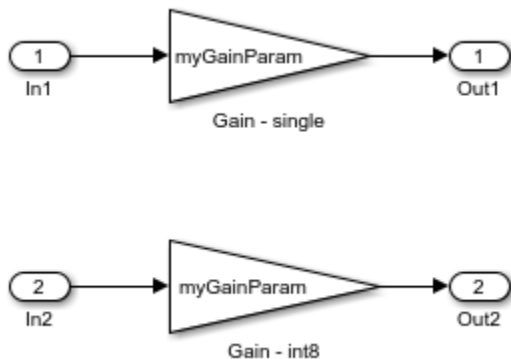
Reuse Parameter Data in Different Data Type Contexts

When you use a `Simulink.Parameter` object or a numeric MATLAB variable to set two or more block parameter values, if the block parameters have different data types, you must explicitly specify the data type of the object or variable. For example, you cannot leave the data type of the parameter object at the default value, `auto`.

Create and Configure Example Model

Create the model `ex_paramdt_contexts`.

`ex_paramdt_contexts`



Set these block parameter values:

- In1 block: **Data type:** `single`
- In2 block: **Data type:** `int8`

In the block labeled `Gain - single`, set these parameter values:

- **Gain:** `myGainParam`
- **Output data type:** `Inherit: Same as input`
- **Parameter data type:** `Inherit: Same as input`

In the block labeled `Gain - int8`, set these parameter values:

- **Gain:** `myGainParam`

- **Output data type:** Inherit: Same as input
- **Parameter data type:** Inherit: Same as input

You can use these commands to set the parameter values:

```
set_param('ex_paramdt_contexts/In1','OutDataTypeStr','single')
set_param('ex_paramdt_contexts/In2','OutDataTypeStr','int8')
set_param('ex_paramdt_contexts/Gain - single','Gain','myGainParam',...
    'OutDataTypeStr','Inherit: Same as input',...
    'ParamDataTypeStr','Inherit: Same as input')
set_param('ex_paramdt_contexts/Gain - int8','Gain','myGainParam',...
    'OutDataTypeStr','Inherit: Same as input',...
    'ParamDataTypeStr','Inherit: Same as input')
```

Create a `Simulink.Parameter` object named `myGainParam` in the base workspace. Configure the object to appear in the generated code as a global variable by applying the storage class `ExportedGlobal`. Configure the object to use the data type `int8`.

```
myGainParam = Simulink.Parameter(3);
myGainParam.CoderInfo.StorageClass = 'ExportedGlobal';
myGainParam.DataType = 'int8';
```

In this model, you use the parameter object `myGainParam` to set two block parameter values. The block parameters inherit different data types from the input signals (`single` or `int8`). To use `myGainParam` in these different data type contexts, explicitly specify the data type of the parameter object by setting the `DataType` property to `int8`.

Match Parameter Object Data Type with Signal Data Type

Optionally, use a `Simulink.NumericType` or `Simulink.AliasType` object to set the parameter object data type and one of the signal data types. This technique eliminates unnecessary typecasts and shifts in the generated code due to a mismatch between the parameter object data type and the signal data type.

Create a `Simulink.NumericType` object to represent the data type `int8`.

```
sharedType_int8 = fixdt('int8');
```

Use this data type object to set:

- The data type of the parameter object. Set the `DataType` property to `sharedType_int8`.
- The data type of the `int8` signal. Use the **Data type** parameter in the Inport block dialog box.

```
myGainParam.DataType = 'sharedType_int8';
set_param('ex_paramdt_contexts/In2', 'OutDataTypeStr', 'sharedType_int8')
```

The parameter object and the signal use the data type `int8`. To change this data type, adjust the properties of the data type object `sharedType_int8`.

Generate and Inspect Code

Generate code from the model.

```
rtwbuild('ex_paramdt_contexts')

### Starting build procedure for model: ex_paramdt_contexts
### Successful completion of build procedure for model: ex_paramdt_contexts
```

The generated file `ex_paramdt_contexts.c` defines the global variable `myGainParam` by using the data type `int8_T`, which corresponds to the data type `int8` in Simulink.

```
file = fullfile('ex_paramdt_contexts_grt_rtw', 'ex_paramdt_contexts.c');
rtwdemodbtype(file, '/* Exported block parameters */', 'int8_T myGainParam = 3;', 1, 1)

/* Exported block parameters */
int8_T myGainParam = 3;          /* Variable: myGainParam
```

The generated code algorithm in the model `step` function uses `myGainParam` to calculate the outputs of the two Gain blocks. In the case of the Gain block whose input signal uses the data type `single`, the code algorithm casts `myGainParam` to the data type `real32_T`, which corresponds to the data type `single` in Simulink.

```
rtwdemodbtype(file, '/* Model step function */', ...
    '/* Model initialize function */', 1, 0)

/* Model step function */
void ex_paramdt_contexts_step(void)
{
    /* Outputport: '<Root>/Out1' incorporates:
     * Gain: '<Root>/Gain - single'
     * Inport: '<Root>/In1'
     */
    ex_paramdt_contexts_Y.Out1 = (real32_T)myGainParam * ex_paramdt_contexts_U.In1;

    /* Outputport: '<Root>/Out2' incorporates:
     * Gain: '<Root>/Gain - int8'
     */
}
```

```
* Inport: '<Root>/In2'  
*/  
ex_paramdt_contexts_Y.Out2 = (int8_T)(myGainParam * ex_paramdt_contexts_U.In2);  
}
```

Related Examples

- “Parameter Data Types in the Generated Code” on page 14-76
- “Generate Efficient Code by Specifying Data Types for Block Parameters” on page 14-81
- “Block Parameter Representation in the Generated Code” on page 14-43

Organize Block Parameter Values into Structures in the Generated Code

In C code, you use structures to store data in contiguous locations in memory. With structures, you can organize data in the code by using meaningful names. Each structure acts as a namespace, so you can reuse a name to represent multiple data items. Similar to arrays, structures enable you to write code that efficiently transfers and operates on large amounts of data by using pointers.

When you configure the generated code to store block parameter values in memory, for example by applying storage classes to `Simulink.Parameter` objects or by creating model arguments, you can organize the parameter data into structures in the generated code.

- Create a parameter structure in Simulink. Use each field of the structure to set a block parameter value. Optionally, use a `Simulink.Bus` object as the data type of the structure. The generated code creates a structure type definition and a structure variable. Use this technique to finely control the data types and order of the fields.
- If you have an Embedded Coder license, apply a custom storage class, such as the built-in custom storage class `Struct`, to multiple `Simulink.Parameter` objects. Use this technique to avoid the effort of creating the structure in Simulink and to organize both signal and parameter data into a single structure. See “Organize Parameter Data into a Structure by Using the Struct Custom Storage Class”.

When you use this technique, to create a nested structure instead of a flat structure, you must create your own custom storage class by writing TLC code. Consider creating a parameter structure in Simulink instead.

Creating Tunable Parameter Structures

When you create a structure in Simulink, you can configure the generated code to define and initialize a corresponding structure variable. Use this technique to share a structure of parameter data with your custom code or to generate parameter data whose values you can change (tune) during code execution.

- 1 Store the structure in a `Simulink.Parameter` object.
- 2 Apply a storage class other than `Auto` to the parameter object.
- 3 Use the fields of the structure to set block parameter values in your model.

The numeric fields of the structure are tunable in the generated code. However, if any field contains a nontunable entity, such as a multidimensional array, none of the structure fields are tunable.

You cannot declare individual substructures or fields within a parameter structure as tunable. You cannot use a `Simulink.Parameter` object as the value of a structure field. Instead, you must store the entire structure in the parameter object.

The fields of parameter structures do not support context-sensitive data typing. However, to match the data type of a field with the data type of another data item in a model, you can use a bus object and a data type object.

- 1 Use a `Simulink.Bus` object as the data type of the structure.
- 2 Use a `Simulink.AliasType` or `Simulink.NumericType` object as the data type of the element in the bus object and as the data type of the target data item.

Control Name and File Placement of Structure Type Definition

When you generate code from a model that uses a tunable parameter structure, by default the code generator creates a `struct` type definition that has a generated name such as `struct_z98c0D2qc4btL`. You can control this type name to improve code readability and integrate the generated code with your own code. Create a `Simulink.Bus` object and use it as the data type of the `Simulink.Parameter` object that stores the structure.

To control the file placement of the generated `struct` type definition, use the `HeaderFile` and `DataScope` properties of the bus object. Use this technique to:

- Export the type definition to a header file that you can include in your custom code.
- Import the type definition from your custom code.

Structures of Parameters

Create a structure in the generated code. The structure stores parameter data.

C Construct

```
typedef struct {  
    double G1;  
    double G2;  
} myStructType;
```

```
myStructType myStruct = {
    2.0,
    -2.0
} ;
```

Procedure

At the command prompt, create a structure named `myStruct` with two fields.

```
myStruct.G1 = 2;
myStruct.G2 = -2;
```

Store the structure in a `Simulink.Parameter` object.

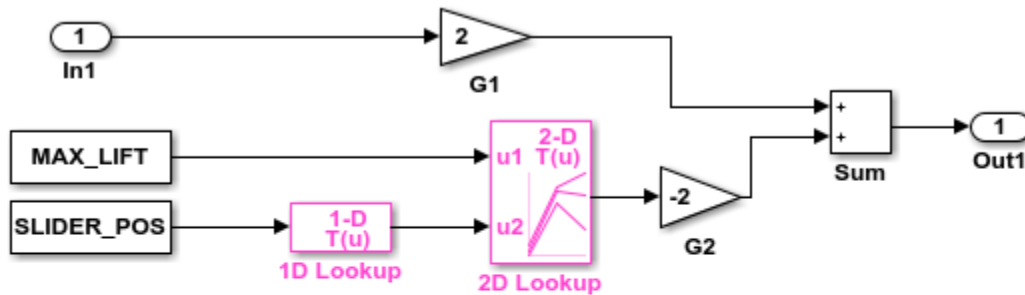
```
myStruct = Simulink.Parameter(myStruct);
```

Apply the storage class `ExportedGlobal` so that the structure appears in the generated code as a global variable.

```
myStruct.CoderInfo.StorageClass = 'ExportedGlobal';
```

Open the example model `rtwdemo_paraminline`.

```
rtwdemo_paraminline
```



<p>Generate Code Using Simulink Coder (double-click)</p>	<p>Generate Code Using Embedded Coder (double-click)</p>	<p>View Optimization Configuration (double-click)</p>	<p>Display Sample Time Colors (double-click)</p>
---	---	--	---

In the G1 block dialog box, set **Gain** to `myStruct.G1`.

```
set_param('rtwdemo_paraminline/G1','Gain','myStruct.G1')
```

In the G2 block dialog box, set **Gain** to `myStruct.G2`.

```
set_param('rtwdemo_paraminline/G2','Gain','myStruct.G2')
```

Results

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for model: rtwdemo_paraminline
### Successful completion of build procedure for model: rtwdemo_paraminline
```

The generated header file `rtwdemo_paraminline_types.h` defines a structure type with a randomized name.

```
file = fullfile('rtwdemo_paraminline_grt_rtw',...
    'rtwdemo_paraminline_types.h');
rtwdemodbtype(file,'typedef struct {'','} struct_6h72eH5WfUEIyQr5YrdGuB;',...
    1,1)
```

```
typedef struct {
    real_T G1;
    real_T G2;
} struct_6h72eH5WfUEIyQr5YrdGuB;
```

The source file `rtwdemo_paraminline.c` defines and initializes the structure variable `myStruct`.

```
file = fullfile('rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.c');
rtwdemodbtype(file,'struct_6h72eH5WfUEIyQr5YrdGuB myStruct',...
    '/* Variable: myStruct',1,1)
```

```
struct_6h72eH5WfUEIyQr5YrdGuB myStruct = {
    2.0,
    -2.0
}; /* Variable: myStruct
```

Specify Name of Structure Type

Optionally, specify a name to use for the structure type definition (struct).

Create a `Simulink.Bus` object that represents the structure type.

```
Simulink.Bus.createObject(myStruct.Value);
```

The default name of the object is `s1Bus1`. Change the name by copying the object into a new MATLAB variable.

```
myStructType = s1Bus1.copy;
```

Use the bus object as the data type of the parameter object.

```
myStruct.DataType = 'Bus: myStructType';
```

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for model: rtwdemo_paraminline  
### Successful completion of build procedure for model: rtwdemo_paraminline
```

The code generates the definition of the structure type `myStructType` and uses this type to define the global variable `myStruct`.

```
rtwdemodbtype(file, 'myStructType myStruct = {' /* Variable: myStruct', ...  
1,1)
```

```
myStructType myStruct = {  
    2.0,  
    -2.0  
}; /* Variable: myStruct
```

Structure Padding

By default, the code generator does not explicitly add padding fields to structure types. Structure types can appear in the generated code through, for example, the default data structures (see “Default Data Structures in the Generated Code” on page 14-16), `Simulink.Bus` objects, and parameter structures that you use in a model.

However, when you use a code replacement library with Embedded Coder, you can specify data alignment (including structure padding) as part of the replacement library. For more information, see “Provide Data Alignment Specifications for Compilers”.

Related Examples

- “Migration to Structure Parameters”
- “Create Tunable Calibration Parameter in the Generated Code” on page 14-56
- “Block Parameter Representation in the Generated Code” on page 14-43
- “Specify Instance-Specific Parameter Values for Reusable Referenced Model” on page 14-61
- “Organize Related Block Parameter Definitions in Structures”

Switch Between Sets of Parameter Values During Simulation and Code Execution

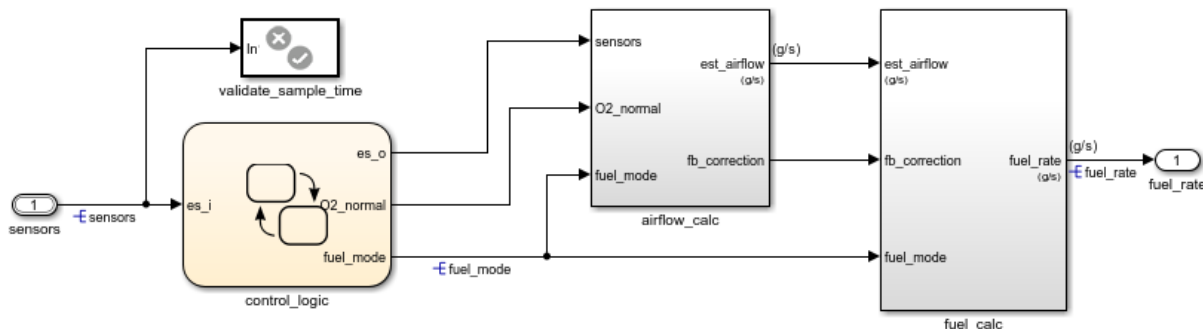
To store multiple independent sets of values for the same block parameters, you can use an array of structures. To switch between the parameter sets, create a variable that acts as an index into the array, and change the value of the variable. You can change the value of the variable during simulation and, if the variable is tunable, during execution of the generated code.

Explore Example Model

Open this example model:

```
open_system('sldemo_fuelsys_dd_controller')
```

Fuel Rate Controller



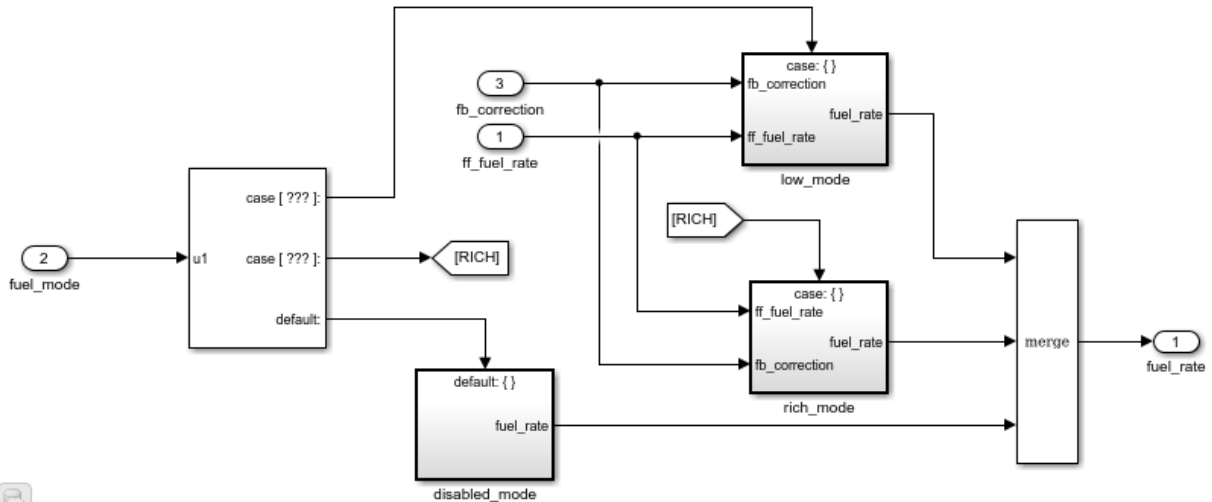
Copyright 1990-2015 The MathWorks, Inc.

This model represents the fueling system of a gasoline engine. The output of the model is the rate of fuel flow to the engine.

Navigate to the `switchable_compensation` nested subsystem.

```
open_system(['sldemo_fuelsys_dd_controller/fuel_calc/', ...
            'switchable_compensation'])
```

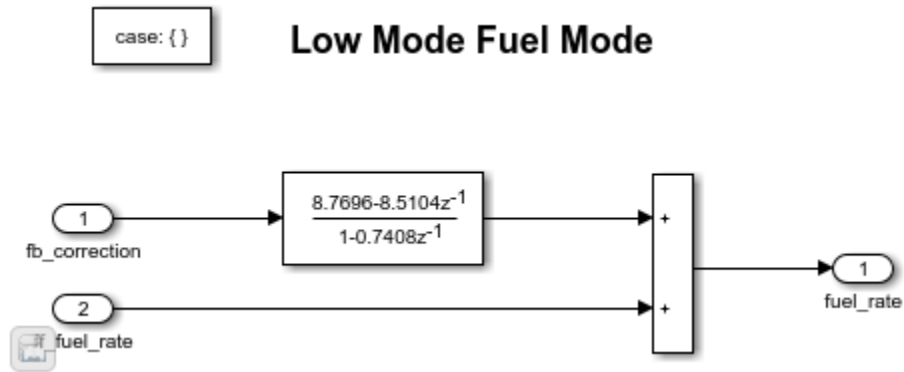
Loop Compensation and Filtering



This subsystem corrects and filters noise out of the fuel rate signal. The subsystem uses different filter coefficients based on the fueling mode, which the control logic changes based on sensor failures in the engine. For example, the control algorithm activates the `low_mode` subsystem during normal operation. It activates the `rich_mode` subsystem in response to sensor failure.

Open the `low_mode` subsystem.

```
open_system(['sldemo_fuelsys_dd_controller/fuel_calc/',...
            'switchable_compensation/low_mode'])
```



The Discrete Filter block filters the fuel rate signal. In the block dialog box, the **Numerator** parameter sets the numerator coefficients of the filter.

The sibling subsystem `rich_mode` also contains a Discrete Filter block, which uses different coefficients.

Update the model diagram to display the signal data types. The input and output signals of the block use the single-precision, floating-point data type `single`.

In the lower-left corner of the model, click the data dictionary badge. The data dictionary for this model, `sldemo_fuelsys_dd_controller.sldd`, opens in the Model Explorer.

In the **Contents** pane, view the properties of the `Simulink.NumericType` objects, such as `s16En15`. All of these objects currently represent the single-precision, floating-point data type `single`. The model uses these objects to set signal data types, including the input and output signals of the Discrete Filter blocks.

Suppose that during simulation and execution of the generated code, you want each of these subsystems to switch between different numerator coefficients based on a variable whose value you control.

Store Parameter Values in Array of Structures

Store the existing set of numerator coefficients in a `Simulink.Parameter` object whose value is a structure. Each field of the structure stores the coefficients for one of the Discrete Filter blocks.

```
lowBlock = ['sldemo_fuelsys_dd_controller/fuel_calc/'...
            'switchable_compensation/low_mode/Discrete Filter'];
```

```
richBlock = ['sldemo_fuelsys_dd_controller/fuel_calc/'...
            'switchable_compensation/rich_mode/Discrete Filter'];
params.lowNumerator = eval(get_param(lowBlock, 'Numerator'));
params.richNumerator = eval(get_param(richBlock, 'Numerator'));
params = Simulink.Parameter(params);
```

Copy the value of `params` into a temporary variable. Modify the field values in this temporary structure, and assign the modified structure as the second element of `params`.

```
temp = params.Value;
temp.lowNumerator = params.Value.lowNumerator * 2;
temp.richNumerator = params.Value.richNumerator * 2;
params.Value(2) = temp;
clear temp
```

The value of `params` is an array of two structures. Each structure stores one set of filter coefficients.

Create Variable to Switch Between Parameter Sets

Create a `Simulink.Parameter` object named `Ctrl`.

```
Ctrl = Simulink.Parameter(2);
Ctrl.DataType = 'uint8';
```

In the `low_mode` subsystem, in the Discrete Filter block dialog box, set the **Numerator** parameter to the expression `params(Ctrl).lowNumerator`.

```
set_param(lowBlock, 'Numerator', 'params(Ctrl).lowNumerator');
```

In the Discrete Filter block in the `rich_mode` subsystem, set the value of the **Numerator** parameter to `params(Ctrl).richNumerator`.

```
set_param(richBlock, 'Numerator', 'params(Ctrl).richNumerator');
```

The expressions select one of the structures in `params` by using the variable `Ctrl`. The expressions then dereference one of the fields in the structure. The field value sets the values of the numerator coefficients.

To switch between the sets of coefficients, you change the value of `Ctrl` to the corresponding index in the array of structures.

Use Bus Object as Data Type of Array of Structures

Optionally, create a `Simulink.BUS` object to use as the data type of the array of structures. You can:

- Control the shape of the structures.
- For each field, control characteristics such as data type and physical units.
- Control the name of the `struct` type in the generated code.

Use the function `Simulink.Bus.createObject` to create the object and rename the object as `paramsType`.

```
Simulink.Bus.createObject(params.Value)
paramsType = slBus1;
clear slBus1
```

You can use the `Simulink.NumericType` objects from the data dictionary to control the data types of the structure fields. In the bus object, use the name of a data type object to set the `DataType` property of each element.

```
paramsType.Elements(1).DataType = 's16En15';
paramsType.Elements(2).DataType = 's16En7';
```

Use the bus object as the data type of the array of structures.

```
params.DataType = 'Bus: paramsType';
```

Use Enumerated Type for Switching Variable

Optionally, use an enumerated type as the data type of the switching variable. You can associate each of the parameter sets with a meaningful name and restrict the allowed values of the switching variable.

Create an enumerated type named `FilterCoeffs`. Create an enumeration member for each of the structures in `params`. Set the underlying integer value of each enumeration member to the corresponding index in `params`.

```
Simulink.defineIntEnumType('FilterCoeffs',{'Weak','Aggressive'},[1 2])
```

Use the enumerated type as the data type of the switching variable. Set the value of the variable to `Aggressive`, which corresponds to the index 2.

```
Ctrl.Value = FilterCoeffs.Aggressive;
```

Add New Objects to Data Dictionary

Add the objects that you created to the data dictionary
`sldemo_fuelsys_dd_controller.slidd`.


```
dictObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
sectObj = getSection(dictObj, 'Design Data');
addEntry(sectObj, 'Ctrl', Ctrl)
addEntry(sectObj, 'params', params)
addEntry(sectObj, 'paramsType', paramsType)
```

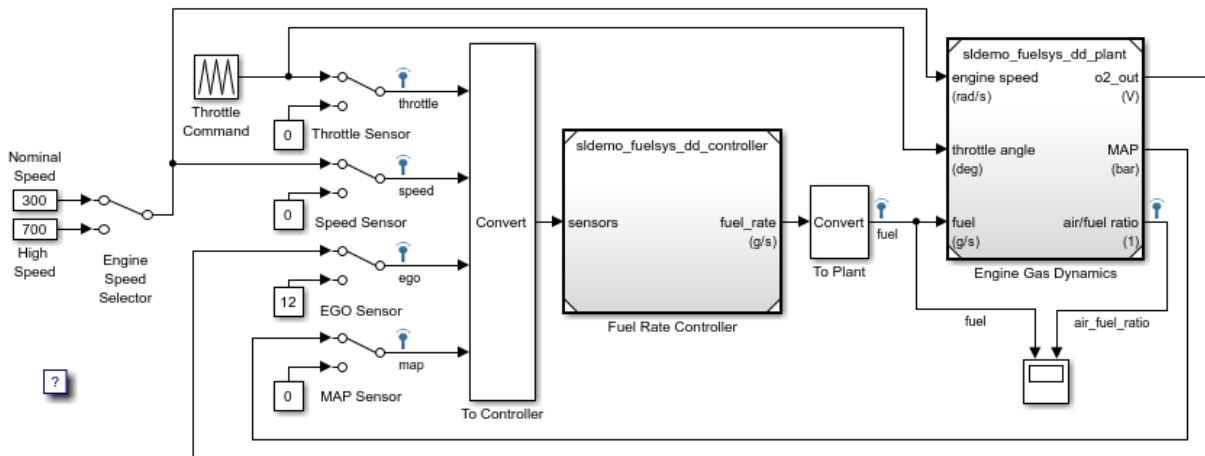
You can also store enumerated types in data dictionaries. However, you cannot import the enumerated type in this case because you cannot save changes to `sldemo_fuelsys_dd_controller.sldd`. For more information about storing enumerated types in data dictionaries, see “Enumerations in Data Dictionary”.

Switch Between Parameter Sets During Simulation

Open the example model `sldemo_fuelsys_dd`, which references the controller model `sldemo_fuelsys_dd_controller`.

```
open_system('sldemo_fuelsys_dd')
```

Fault-Tolerant Fuel Control System



The sensor switches simulate any combination of sensor failures.
The Engine Speed Selector switch simulates different engine speeds (rad/s).

Copyright 1990-2015 The MathWorks, Inc.

Set the simulation stop time to `Inf` so that you can interact with the model during simulation.

Begin a simulation run and open the Scope block dialog box. The scope shows that the fuel flow rate (the `fuel` signal) oscillates with significant amplitude during normal operation of the engine.

In the Model Explorer, view the contents of the data dictionary `sldemo_fuelsys_dd_controller.sldd`. Set the value of `Ctrl` to `FilterCoeffs.Weak`.

Update the `sldemo_fuelsys_dd` model diagram. The scope shows that the amplitude of the fuel rate oscillations decreases due to the less aggressive filter coefficients.

Stop the simulation.

Generate and Inspect Code

If you have Simulink Coder software, you can generate code that enables you to switch between the parameter sets during code execution.

In the Model Explorer, view the contents of the data dictionary `sldemo_fuelsys_dd_controller.sldd`. In the **Contents** pane, set **Column View** to **Storage Class**.

Use the **StorageClass** column to apply the storage class `ExportedGlobal` to `params` so that the array of structures appears as a tunable global variable in the generated code. Apply the same storage class to `Ctrl` so that you can change the value of the switching variable during code execution.

Alternatively, to configure the objects, use these commands:

```
tempEntryObj = getEntry(sectObj, 'params');
params = getValue(tempEntryObj);
params.StorageClass = 'ExportedGlobal';
setValue(tempEntryObj, params);

tempEntryObj = getEntry(sectObj, 'Ctrl');
Ctrl = getValue(tempEntryObj);
Ctrl.StorageClass = 'ExportedGlobal';
setValue(tempEntryObj, Ctrl);
```

Generate code from the controller model.

```
rtwbuild('sldemo_fuelsys_dd_controller')

### Starting build procedure for model: sldemo_fuelsys_dd_controller
```

```
### Successful completion of code generation for model: sldemo_fuelsys_dd_controller
```

In the code generation report, view the header file `sldemo_fuelsys_dd_controller_types.h`. The code defines the enumerated data type `FilterCoeffs`.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw',...
    'sldemo_fuelsys_dd_controller_types.h');
rtwdemodbtype(file, '#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_',...
    '/* Forward declaration for rtModel */',1,0)
```

```
#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_
#define DEFINED_TYPEDEF_FOR_FilterCoeffs_

typedef enum {
    Weak = 1,                /* Default value */
    Aggressive
} FilterCoeffs;

#endif
```

The code also defines the structure type `paramType`, which corresponds to the `Simulink.Bus` object. The fields use the single-precision, floating-point data type from the model.

```
rtwdemodbtype(file, '#ifndef DEFINED_TYPEDEF_FOR_paramType_',...
    '#ifndef DEFINED_TYPEDEF_FOR_FilterCoeffs_',1,0)
```

```
#ifndef DEFINED_TYPEDEF_FOR_paramType_
#define DEFINED_TYPEDEF_FOR_paramType_

typedef struct {
    real32_T lowNumerator[2];
    real32_T richNumerator[2];
} paramType;

#endif
```

View the source file `sldemo_fuelsys_dd_controller.c`. The code uses the enumerated type to define the switching variable `Ctrl`.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw',...
    'sldemo_fuelsys_dd_controller.c');
```

```
rtwdemodbtype(file,'FilterCoeffs Ctrl = Aggressive;',...
    '/* Block signals (auto storage) */',1,0)

FilterCoeffs Ctrl = Aggressive;          /* Variable: Ctrl
                                           * Referenced by:
                                           *   '<S12>/Discrete Filter'
                                           *   '<S13>/Discrete Filter'
                                           */
```

The code also defines the array of structures `params`.

```
rtwdemodbtype(file,'/* Exported block parameters */',...
    '/* Variable: params',1,1)

/* Exported block parameters */
paramsType params[2] = { {
    { 8.7696F, -8.5104F },

    { 0.0F, 0.2592F }
}, { { 17.5392F, -17.0208F },

    { 0.0F, 0.5184F }
} } ;                                     /* Variable: params
```

The code algorithm in the model `step` function uses the switching variable to index into the array of structures.

To switch between the parameter sets stored in the array of structures, change the value of `Ctrl` during code execution.

Related Examples

- “Tune and Experiment with Block Parameter Values”
- “Block Parameter Representation in the Generated Code” on page 14-43
- “Organize Related Block Parameter Definitions in Structures”

Signal Representation in Generated Code

In this section...

“Signal Storage Concepts” on page 14-110

“Signals with Auto Storage Class” on page 14-112

“Signals with Test Points” on page 14-114

“Symbolic Naming Conventions for Signals” on page 14-115

“Summary of Signal Storage Class Options” on page 14-116

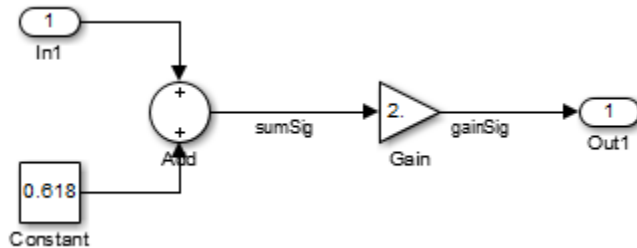
“Interfaces for Monitoring Signals” on page 14-117

“Share Data Between Code Generated from Simulink, Stateflow, and MATLAB” on page 14-117

The code generator offers a number of options that let you control how signals in your model are stored and represented in the generated code.

- Control whether signal storage is declared in global memory space or locally in functions (that is, in stack variables).
- Control the allocation of stack space when using local storage.
- Declare signals as *test points* to store them in unique memory locations
- Reduce memory usage by instructing the code generator to store signals in reusable buffers.
- Control whether or not signals declared in generated code are interfaceable (visible) to externally written code. You can also specify that signals are to be stored in locations declared by externally written code.
- Preserve the symbolic names of signals in generated code by using signal labels.

The discussion in the following sections refers to code generated from `signal_examp`, the model shown in the next figure.



signal_examp Model

Signal Storage Concepts

This section discusses structures and concepts you must understand to choose the best signal storage options for your application:

- The global block I/O data structure `model_B`
- The concept of signal *storage classes* as used by the code generator

Global Block I/O Structure

By default, the code generator attempts to optimize memory usage by sharing signal memory and using local variables.

However, under a number of circumstances you should place signals in global memory. For example,

- You might want a signal to be stored in a structure that is visible to externally written code.
- The number and/or size of signals in your model might exceed the stack space available for local variables.

In such cases, it is possible to override the default behavior and store selected signals in a model-specific *global block I/O data structure*. The global block I/O structure is called `model_B` (in earlier versions this was called `rtB`).

The following code shows how `model_B` is defined and declared in code generated (with signal storage optimizations off) from the `signal_examp` model shown in the `signal_examp` Model figure.

```
(in signal_examp.h)
/* Block signals (auto storage) */
extern B_signal_examp_T signal_examp_B;
```

```
(in signal_examp.c)
/* Block signals (auto storage) */
B_signal_examp_T signal_examp_B;
```

Field names for signals stored in *model_B* are generated according to the rules described in “Symbolic Naming Conventions for Signals” on page 14-115.

In certain cases, the code generator places signals in the block I/O structure, even when you specify the storage class for the signal object as **Auto** or if you enable the option to reuse the signal. This override occurs when the values of these signals need to be persistent across time step. In such cases, the only way to represent these signals locally in generated code is to change the semantics of your Simulink model.

Signal Storage Class

The *storage class* property of a signal specifies how the product declares and stores the signal. In some cases this specification is qualified by more options.

In the context of the code generator, the term “storage class” is not synonymous with the term *storage class specifier*, as used in the C language.

Default Storage Class

Auto is the default storage class and is the storage class you should use for signals that you do not need to interface to external code. Signals with **Auto** storage class can be stored in local and/or shared variables or in a global data structure. The form of storage depends on the **Signal storage reuse**, **Reuse local block outputs**, and **Enable local block outputs** options, and on available stack space. See “Signals with Auto Storage Class” on page 14-112 for a full description of code generation options for signals with **Auto** storage class.

Explicitly Assigned Storage Classes

Signals with storage classes other than **Auto** are stored either as members of an appropriate global data structure, such as *model_B* or *model_U*, or in unstructured global variables, independent of the global data structures. These storage classes are for signals that you want to monitor and/or interface to external code.

The **Signal storage reuse**, **Enable local block outputs**, **Reuse local block outputs**, and **Eliminate superfluous local variables (expression folding)** optimizations do not apply to signals with storage classes other than `Auto`.

Use the Signal Properties dialog box to assign these storage classes to signals:

- **SimulinkGlobal**: The signal is stored as a field of a global data structure such as the block I/O structure. Signals with `SimulinkGlobal` storage class must have unique signal names. See “Control Signals and States in Code by Applying Storage Classes” on page 14-120 for more information.
- **ExportedGlobal**: The signal is stored in a global variable, independent of the global data structures. `model.h` exports the variable. Signals with `ExportedGlobal` storage class must have unique signal names. See “Control Signals and States in Code by Applying Storage Classes” on page 14-120 for more information.
- **ImportedExtern**: `model_private.h` declares the signal as an `extern` variable. Your code must supply the variable definition. Signals with `ImportedExtern` storage class must have unique signal names. See “Control Signals and States in Code by Applying Storage Classes” on page 14-120 for more information.
- **ImportedExternPointer**: `model_private.h` declares the signal as an `extern` pointer. Your code must define a valid pointer variable. Signals with `ImportedExtern` storage class must have unique signal names. See “Control Signals and States in Code by Applying Storage Classes” on page 14-120 for more information.

Signals with Auto Storage Class

Options are available for signals with `Auto` storage class:

- **Signal storage reuse**
- **Enable local block outputs**
- **Reuse local block outputs**
- **Eliminate superfluous local variables (expression folding)**

Use these options to control signal memory reuse and choose local or global (`model_B`) storage for signals. These options are on the **All Parameters** tab of the Configuration Parameters dialog box.

These options interact. When the **Signal storage reuse** option is selected,

- The **Reuse local block outputs** option is enabled and selected, and signal memory is reused whenever possible, reducing stack size where signals are being buffered in local variables.
- The **Enable local block outputs** option is enabled and selected. This parameter lets you choose whether reusable signal variables are declared as local variables in functions or as members of *model_B*.
- The **Eliminate superfluous local variables (expression folding)** is enabled and selected, and block computations collapse into single expressions.

The following code examples illustrate the effects of the **Signal storage reuse**, **Enable local block outputs**, and **Reuse local block outputs** options. The examples were generated from the `signal_examp` model (see figure `signal_examp Model`). For clarity in showing the individual Gain and Sum block computation, expression folding is off in this example.

This code example shows signal storage optimization, with **Signal storage reuse**, **Enable local block outputs**, and **Reuse local block outputs** selected. The local variable `rtb_gainSig` holds the outputs of the Sum and Gain blocks.

```
/* Model step function */
void signal_examp_step(void)
{
    real_T rtb_gainSig;

    /* Sum: '<Root>/Sum' incorporates:
     * Constant: '<Root>/Constant'
     * Inport: '<Root>/In1'
     */
    rtb_gainSig = signal_examp_U.In1 + signal_examp_P.Constant_Value;

    /* Gain: '<Root>/Gain' */
    rtb_gainSig *= signal_examp_P.Gain_Gain;

    /* Outport: '<Root>/Out1' */
    signal_examp_Y.Out1 = rtb_gainSig;
}
```

This example shows the code with **Signal storage reuse** cleared. The global variable `signal_example_B.sumSig` holds the output of the Sum block.

```
/* Model step function */
void signal_examp_step(void)
{
    /* Sum: '<Root>/Sum' incorporates:
     * Constant: '<Root>/Constant'
     * Inport: '<Root>/In1'
     */
    /* ... */
}
```

```

signal_examp_B.sumSig = signal_examp_P.Constant_Value + signal_examp_U.In1;

/* Output: '<Root>/Out1' incorporates:
 * Gain: '<Root>/Gain'
 */
signal_examp_Y.Out1 = signal_examp_P.Gain_Gain * signal_examp_B.sumSig;
}

```

In large models, disabling **Signal storage reuse** can significantly increase RAM and ROM usage. Therefore, this approach is not recommended for code deployment; however it can be useful in rapid prototyping environments.

The following table summarizes the possible combinations of the **Signal storage reuse / Reuse block outputs** and **Enable local block outputs** options.

	Signal storage reuse and Reuse local block outputs ON	Signal storage reuse OFF (Reuse local block outputs disabled)
Enable local block outputs ON	Reuse signals in local memory (fully optimized)	N/A
Enable local block outputs OFF	Reuse signals in <i>model_B</i> structure	Individual signal storage in <i>model_B</i> structure

Signals with Test Points

A *test point* is a signal that is stored in a unique location that other signals cannot share or reuse. See “Test Points” in the Simulink documentation for information about including test points in your model.

When you generate code for models that include test points, the build process allocates a separate memory buffer for each test point. Test points are stored as members of an appropriate global data structure such as *model_B* or *model_U*.

Declaring a signal as a test point disables the following options for that signal. This can lead to increased code and data size. You do not lose the benefits of optimized storage for other signals in your model.

- **Signal storage reuse**
- **Enable local block outputs**
- **Reuse local block outputs**

- **Eliminate superfluous local variables (expression folding)**

For an example of storage declarations and code generated for a test point, see “Summary of Signal Storage Class Options” on page 14-116.

If you have an Embedded Coder license, you can specify that the build process ignore test points in the model, allowing optimal buffer allocation, using the “Ignore test point signals” parameter. Ignoring test points facilitates transitioning from prototyping to deployment and avoids accidental degradation of generated code due to workflow artifacts. For more information, see “Ignore test point signals”.

Symbolic Naming Conventions for Signals

When signals have a storage class other than `Auto`, the code generator preserves symbolic information about the signals or their originating blocks in the generated code.

For labeled signals, field names in `model_B` derive from the signal names. In the following example, the field names `model_B.sumSig` and `model_B.gainSig` are derived from the corresponding labeled signals in the `signal_examp` model (shown in figure `signal_examp Model`).

```
/* Block signals (auto storage) */
typedef struct _BlockIO_signal_examp {
    real_T sumSig;                /* '<Root>/Add' */
    real_T gainSig;              /* '<Root>/Gain' */
} BlockIO_signal_examp;
```

When you clear the **Signal storage reuse** optimization, `sumSig` is not part of `model_B`, and a local variable is used for it instead. For unlabeled signals, `model_B` field names are derived from the name of the source block or subsystem.

The components of a generated signal label are

- The root model name, followed by
- The name of the generating signal object, followed by
- Unique *name-mangling* text (if required)

The number of characters that a signal label can have is limited by the **Maximum identifier length** parameter specified on the **Symbols** pane of the Configuration Parameters dialog box. See “Construction of Generated Identifiers” on page 16-14 for more detail.

When a signal has **Auto** storage class, the build process controls generation of variable or field names without regard to signal labels.

Summary of Signal Storage Class Options

The next table shows, for each signal storage class option, the variable declaration and the code generated for Sum (**sumSig**) and Gain (**gainSig**) block outputs of the model shown in figure `signal_examp` Model.

Storage Class	Declaration	Code
Auto (with Signal storage reuse optimizations on)	In <code>model.c</code> or <code>model.cpp</code> <code>real_T rtb_sumSig;</code>	<code>rtb_sumSig = signal_examp_U.In1 + signal_examp_P.Constant_Value; rtb_sumSig *= signal_examp_P.Gain_Gain; signal_examp_Y.Out1 = rtb_sumSig;</code>
Test point (for <code>sumSig</code> only)	In <code>model.h</code> <code>typedef struct _BlockIO_signal_examp { real_T sumSig; } BlockIO_signal_examp;</code> In <code>model.c</code> or <code>model.cpp</code> <code>BlockIO_signal_examp signal_examp_B; real_T rtb_gainSig;</code>	<code>signal_examp_B.sumSig = signal_examp_U.In1 + signal_examp_P.Constant_Value; rtb_gainSig = signal_examp_B.sumSig * signal_examp_P.Gain_Gain; signal_examp_Y.Out1 = rtb_gainSig;</code>
ExportedGlobal (for <code>sumSig</code> only)	In <code>model.h</code> <code>extern real_T sumSig;</code> In <code>model.c</code> or <code>model.cpp</code> <code>real_T sumSig; real_T rtb_gainSig;</code>	<code>sumSig = signal_examp_U.In1 + signal_examp_P.Constant_Value; rtb_gainSig = sumSig * signal_examp_P.Gain_Gain; signal_examp_Y.Out1 = rtb_gainSig;</code>
ImportedExtern	In <code>model_private.h</code> <code>extern real_T sumSig;</code> In <code>model.c</code> or <code>model.cpp</code> <code>real_T rtb_gainSig;</code>	<code>sumSig = signal_examp_U.In1 + signal_examp_P.Constant_Value; rtb_gainSig = sumSig * signal_examp_P.Gain_Gain; signal_examp_Y.Out1 = rtb_gainSig;</code>
ImportedExternPointer	In <code>model_private.h</code>	<code>(*sumSig) = signal_examp_U.In1 + signal_examp_P.Constant_Value;</code>

Storage Class	Declaration	Code
	<pre>extern real_T *sumSig;</pre> <p>In <i>model.c</i> or <i>model.cpp</i></p> <pre>real_T rtb_gainSig;</pre>	<pre>rtb_gainSig = (*sumSig) * signal_examp_P.Gain_Gain; signal_examp_Y.Out1 = rtb_gainSig;</pre>

Interfaces for Monitoring Signals

The code generator includes

- Support for developing a Target Language Compiler API for monitoring signals and states independent of external mode. See “Input Signal Functions” and “Output Signal Functions” in the Target Language Compiler documentation for information.
- A C application program interface (API) for monitoring signals and states independent of external mode. See “Exchange Data Between Generated and External Code Using C API” on page 28-2 for information.
- An interface for exporting ASAP2 files, which you customize to use signal objects. For details, see “Export ASAP2 File for Data Measurement and Calibration” on page 29-2.

Share Data Between Code Generated from Simulink, Stateflow, and MATLAB

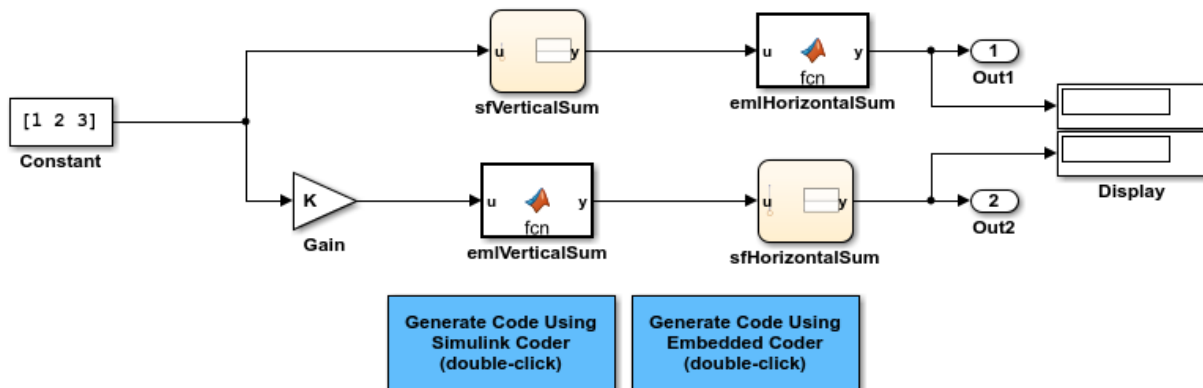
Stateflow and MATLAB Coder can fully define their data definitions, or they can inherit them from Simulink. Data definition capabilities include:

- Inheriting input/output data types and sizes from Simulink.
- Parameterized data types and sizes. That is, data type and size may be specified as a function of another data's type and size, e.g., `type(y)=type(u)` and `size(y)=size(u)`.
- Inferred output size and type from Simulink via signal attribute back propagation.
- Parameter scoped data, which allows referencing Simulink parameters in Stateflow and MATLAB.

Open Example Model

Open the example model `rtwdemo_dynamicio`.

```
open_system('rtwdemo_dynamicio')
```



Copyright 1994-2015 The MathWorks, Inc.

Instructions

- 1 Compile the model (**Simulation > Update Diagram**) and note the displayed signal types and sizes.
- 2 Change the data type and/or size of the Constant block and recompile the model. Note that the attributes of the signals automatically adapt to the Constant block specification.
- 3 Generate and inspect code using the blue buttons in the model. Note that K is shared by the Gain and sfVerticalSum blocks.

Notes

- The data type and size of all Stateflow and MATLAB data is inherited from Simulink.
- The Gain block and the Stateflow chart sfVerticalSum share the Simulink parameter K, which is defined in the MATLAB workspace as a Simulink.Parameter with SimulinkGlobal storage class (i.e., rtP.K in the generated code).

Related Examples

- “Control Signals and States in Code by Applying Storage Classes” on page 14-120

- “Control Signal and State Initialization in the Generated Code” on page 14-145
- “Maximize Signal Storage Optimization” on page 14-144
- “Default Data Structures in the Generated Code” on page 14-16
- “Storage Classes for Signals Used with Model Blocks” on page 8-22
- “Control Signal Data Types”

Control Signals and States in Code by Applying Storage Classes

Signals and block states appear in the generated code as variables. For basic information about signal representation in the generated code, see “Signal Representation in Generated Code” on page 14-109. For basic information about state representation, see “Discrete Block State Naming in Generated Code” on page 14-158 and “Continuous Block State Naming in Generated Code” on page 14-156.

In a model, you can assign each signal and state a *storage class* to determine the variable scope in the generated code. You can use storage classes to monitor signal and state data during execution and interface signals and states to externally written code.

To specify storage classes for signals and states, you can use *signal objects*, which you store in a workspace or data dictionary. Signal objects are objects of the class `Simulink.Signal`. If you create your own data class package, signal objects are also objects of the subclass of `Simulink.Signal` that your package defines. For basic information about data objects, see “Data Objects” in the Simulink documentation.

You can apply storage classes directly to signal lines and block states by using the Signal Properties dialog box or the **State Attributes** tab of a block dialog box. This technique does not require you to store a signal object in a workspace.

You can interface test points and other signals that are stored as members of `model_B`, or of another global data structure such as `model_U`, to your code. To do this, your code must know the address of the global data structure where the data is stored, and other information. This information is not automatically exported. The code generator provides C/C++ and Target Language Compiler APIs that give your code access to `model_B` and other data structures. See “Interfaces for Monitoring Signals” on page 14-117 for more information.

If you have an Embedded Coder license, you can use and create custom storage classes to represent more complex data such as structures and macros. You can also use custom storage classes to export data declarations to specific generated files. For more information, see “Introduction to Custom Storage Classes” and “Simulink Package Custom Storage Classes”.

In this section...

“Storage Classes for Signals and States” on page 14-121

“Use Model Data Editor to Configure Data Interface” on page 14-124

“Signal Objects for Code Generation” on page 14-125

In this section...

“Create and Configure Signal Object for Code Generation” on page 14-125

“Programmatically Create and Configure Signal Object for Code Generation” on page 14-126

“Apply Storage Classes Directly to Signal Lines, Block States, and Output Blocks” on page 14-126

“Programmatically Apply Storage Classes Directly to Signals, States, and Output Blocks” on page 14-127

“Resolve Conflicts in Configuration of Signal Objects” on page 14-128

Storage Classes for Signals and States

To control the code generated for signals and states, use storage class. For example, you can import or export the corresponding variable to or from the code.

You can choose from these built-in storage classes:

- **Auto**, which is the default storage class. The code generator determines signal or state storage based on optimization settings such as configuration parameters.
- **SimulinkGlobal**. The generated code contains a structured global variable for signals and another structured global variable for states that use this storage class. Each signal or state appears as a field of the appropriate structure.
- **ExportedGlobal**. Export the signal or state as a unique global variable in the generated code. The code contains an **extern** declaration for the variable.
- **ImportedExtern**. Import the signal or state as a unique global variable in the generated code. Your code must provide the variable definition.
- **ImportedExternPointer**. Import the signal or state as a unique global pointer variable in the generated code. Your code must provide the pointer variable definition.

Storage Classes for Signal Lines

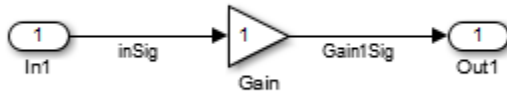
To control the code generated for signals that use **Auto** storage class, on the **All Parameters** tab of the Configuration Parameters dialog box, adjust these options:

- **Signal storage reuse**
- **Reuse block outputs**
- **Enable local block outputs**

• **Eliminate superfluous local variables (expression folding)**

These configuration parameters determine, for example, whether a signal appears in the code as a local variable or whether expression folding eliminates the signal altogether.

To customize the code generation for an individual signal, specify a storage class other than `Auto`. For each of the storage classes, the table shows the variable declaration and the code generated for the Inport signal, `inSig`, of the example model `signal_examp`.



Storage Class	Declaration and Definition	Code
Auto (with storage optimizations on)	In <code>signal_examp.h</code> <pre>typedef struct { real_T inSig; } ExtU_signal_examp_T; extern ExtU_signal_examp_T signal_examp_U;</pre>	<code>Gain1Sig = signal_examp_P.Gain_Gain * signal_examp_U.inSig;</code>
SimulinkGlobal	In <code>signal_examp.h</code> <pre>typedef struct { real_T inSig; } ExtU_signal_examp_T; extern ExtU_signal_examp_T signal_examp_U;</pre>	<code>Gain1Sig = signal_examp_P.Gain_Gain * signal_examp_U.inSig;</code>
ExportedGlobal	In <code>signal_examp.c</code> <pre>real_T inSig;</pre> In <code>signal_examp.h</code> <pre>extern real_T inSig;</pre>	<code>Gain1Sig = inSig * signal_examp_P.Gain_Gain;</code>
ImportedExtern	In <code>signal_examp_private.h</code> <pre>extern real_T inSig;</pre>	<code>Gain1Sig = inSig * signal_examp_P.Gain_Gain;</code>
ImportedExternPointer	In <code>signal_examp_private.h</code>	<code>Gain1Sig = (*inSig) * signal_examp_P.Gain_Gain;</code>

Storage Class	Declaration and Definition	Code
	<code>extern real_T *inSig;</code>	

Storage Classes for Block States

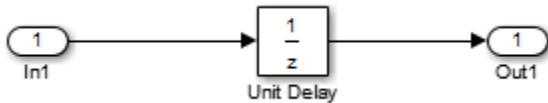
Use the `Auto` storage class for states that you do not need to interface to external code. States with `Auto` storage class are typically stored as fields of the `DWork` structure in the generated code.

You can assign a symbolic name to states that use the `Auto` storage class. If you do not supply a state name, the code generator produces one, as described in “Discrete Block State Naming in Generated Code” on page 14-158.

States with `SimulinkGlobal` storage class are stored as fields of the `DWork` structure.

Block states with storage classes other than `Auto` or `SimulinkGlobal` are stored in unstructured global variables, independent of the `DWork` vector. Use these storage classes for states that you want to interface to external code.

For each of the storage classes, the table shows the variable declaration and initialization code generated for the Unit Delay block state in the example model `state_examp`. The block state name is specified as `udx`. The initial value is 0.



Storage Class	Declaration and Definition	Initialization Code
<code>Auto</code>	In <code>state_examp.h</code> <pre>typedef struct { real_T udx; } DW_state_examp_T;</pre>	<code>state_examp_DW.udx = 0.0;</code>
<code>SimulinkGlobal</code>	In <code>state_examp.h</code> <pre>typedef struct { real_T udx; } DW_state_examp_T;</pre>	<code>state_examp_DW.udx = 0.0;</code>

Storage Class	Declaration and Definition	Initialization Code
ExportedGlobal	In <code>state_examp.c</code> <code>real_T udx;</code> In <code>state_examp.h</code> <code>extern real_T udx;</code>	<code>udx = 0.0;</code>
ImportedExtern	In <code>state_examp_private.h</code> <code>extern real_T udx;</code>	<code>udx = 0.0;</code>
ImportedExternPointer	In <code>state_examp_private.h</code> <code>extern real_T *udx;</code>	<code>*udx = state_examp_U.In1;</code>

Use Model Data Editor to Configure Data Interface

Use the Model Data Editor to apply storage classes to Inport and Outport blocks, signal lines, and Data Store Memory blocks. Use this technique to apply the storage classes without locating the blocks and signals in the model and to configure the data interface of the model by using a single list.

To open the Model Data Editor, select **View > Model Data**. For information about using the Model Data Editor, see “Configure Data Properties by Using a Table”. For an example that shows how to apply storage classes to Inport and Outport blocks, see “Design Data Interface by Configuring Inport and Outport Blocks” on page 14-131.

- For an Inport block, the Model Data Editor applies the storage class to the output signal of the block, not to the block itself.
- You can use the Model Data Editor to apply a storage class directly to a root-level Outport block or to the input signal that drives the block.
 - To store the storage class specification in the Outport block, use the **Inports/Outports** tab in the Model Data Editor. When you use this technique, the specification remains after you delete the input signal that drives the block. Use this technique to configure the model interface before you develop the internal algorithm.
 - To store the specification in the input signal that drives the block, use the **Signals** tab in the Model Data Editor or use the Signal Properties dialog box. Use this technique to use the custom storage class `Reusable` (Embedded Coder). To use

this custom storage class, you associate the signal line with a `Simulink.Signal` object in a workspace or data dictionary instead of storing the specification in the signal line.

Signal Objects for Code Generation

To specify code generation options for a signal or state in a model by using a data object:


- 1 Create a signal data object, which is an object of the class `Simulink.Signal` or of a subclass of `Simulink.Signal`.
- 2 Specify code generation options by modifying the `CoderInfo` property of the signal object.
- 3 Associate the signal object with a signal or state in a model diagram. For example, in a Signal Properties dialog box, or the **State Attributes** tab of a block dialog box, specify the object name as the name of the target signal or state.
- 4 Generate code and build your target executable.

Signal objects have a property `CoderInfo` that contains an object of the class `Simulink.CoderInfo`. You can use the properties of the `Simulink.CoderInfo` object to specify code generation options for the target model signal or state.

You cannot resolve an Outputport block to a `Simulink.Signal` object. However, you can resolve the input signal of the block to a signal object.

Create and Configure Signal Object for Code Generation

To control the code generation of a signal by creating a signal object, you can use Model Explorer.

- 1 In the Model Explorer **Model Hierarchy** pane, select a workspace to contain the signal object. For example, select `Base Workspace`.
- 2 Click Add Signal .

A `Simulink.Signal` object named `Sig` appears in the base workspace.
- 3 In the **Contents** pane, change the name of the signal object to a meaningful name. For example, name the object `mySignal` or `myState`.
- 4 Select the signal object in the **Contents** pane. In the **Dialog** pane, in the **Storage class** drop-down list, select a storage class such as `ExportedGlobal`. Click **Apply**.

- 5 Open the Signal Properties dialog box for a signal in a model, or open the **State Attributes** tab of a block dialog box. Specify **Signal name** or **State name** as `mySignal` or `myState` and click **Apply**.
- 6 Select the check box next to **Signal name must resolve to Simulink signal object** or **State name must resolve to Simulink signal object**. Click **OK**.
- 7 Generate code.

Programmatically Create and Configure Signal Object for Code Generation

At the command prompt, you can control the code generation of a signal by creating a signal object.

- 1 Create a `Simulink.Signal` object named `mySignal` or `myState` in the base workspace.

```
mySignal = Simulink.Signal;
```

- 2 Specify a storage class for the object. For example, specify the storage class `ExportedGlobal`.

```
mySignal.StorageClass = 'ExportedGlobal';
```

The `StorageClass` property is a property of the `Simulink.CoderInfo` object that resides in the `CoderInfo` property of the signal object. However, you can use the preceding syntax to access the storage class property. You can also explicitly access the property by using the syntax `mySignal.CoderInfo.StorageClass`.

- 3 Open the Signal Properties dialog box for a signal in a model, or open the **State Attributes** tab of a block dialog box. Specify **Signal name** or **State name** as `mySignal` or `myState` and click **Apply**.
- 4 Select the check box next to **Signal name must resolve to Simulink signal object** or **State name must resolve to Simulink signal object**. Click **OK**.
- 5 Generate code.

Apply Storage Classes Directly to Signal Lines, Block States, and Output Blocks

Through dialog boxes, you can apply storage classes directly to signal lines and block states. You do not need a data object that you store in a workspace or data dictionary.

However, if you specify a storage class for a signal or state with this technique, you cannot use a signal object in a workspace to specify other characteristics of the signal or state, such as data type.

To apply a storage class directly to a signal line, use the Signal Properties dialog box. For a block state, use the **State Attributes** tab in the block dialog box.

- 1 Open the **Code Generation** tab in a Signal Properties dialog box, or the **State Attributes** tab in a block dialog box.
- 2 Specify a name in the **Signal name** box or the **State name** box. Click **Apply**.
- 3 In the **Storage class** drop-down list, select a storage class.

To apply a storage class directly to an Outport block, use the Model Data Editor. You can also use the Model Data Editor to apply custom storage classes to signals through a list that you can sort, group, and filter. See “Use Model Data Editor to Configure Data Interface” on page 14-124.

Programmatically Apply Storage Classes Directly to Signals, States, and Outport Blocks

To programmatically apply storage classes to signal lines and block states, use the function `set_param`. You can apply a storage class without creating a `Simulink.Signal` object in a workspace or data dictionary. The storage class specification is saved in the model file.

However, you can use this technique to specify only a storage class for the object. You must specify other signal or state characteristics, such as data type, in the source block dialog box. You cannot use a signal object in a workspace to specify these other characteristics.

This example shows how to programmatically apply a storage class to a signal line.

- 1 Open the example model `rtwdemo_secondOrderSystem`.

```
rtwdemo_secondOrderSystem
```

- 2 Get a handle to the output of the block named Force: `f(t)`.

```
portHandles = get_param('rtwdemo_secondOrderSystem/Force: f(t)', 'PortHandles');  
outportHandle = portHandles.Outport;
```

- 3 Set the name of the corresponding signal to `ForceSignal`.

```
set_param(outportHandle, 'Name', 'ForceSignal')
```

- 4 Set the storage class of the signal to `ExportedGlobal`.

```
set_param(outportHandle, 'StorageClass', 'ExportedGlobal')
```

- 5 Generate code from the model. The code declares and defines a global variable `ForceSignal` to represent the signal.

To apply a storage class directly to an `Outport` block, using the function `set_param`, specify the block parameter `SignalName` to name the signal that the block represents. Use the parameter `StorageClass` to specify a storage class.

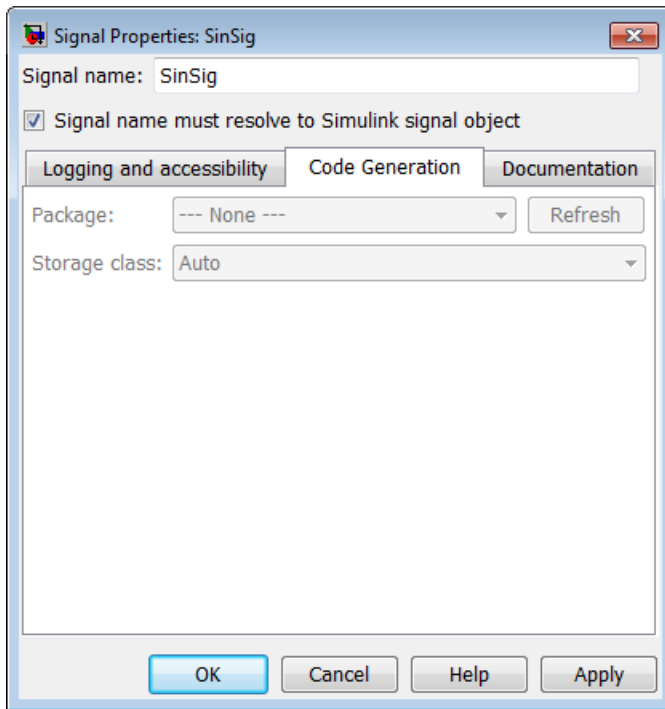
To apply a storage class to a block state, using the function `set_param`, specify the block parameter `StateIdentifier` to name the state. Use the parameter `StateStorageClass` to specify a storage class.

To apply a storage class to a data store that you define by using a `Data Store Memory` block, use the block parameter `StateStorageClass`. You do not need to specify a state name because the data store already has a name.

To programmatically apply a custom storage class to a signal, state, `Outport` block, or data store, use embedded signal objects. Custom storage classes do not affect the generated code unless you use a system target file based on `ert.tlc`, which requires Embedded Coder. You can also use this technique to set storage classes. See “Programmatically Apply Custom Storage Classes Directly to Signals, States, and `Outport` Blocks Using Embedded Signal Objects”.

Resolve Conflicts in Configuration of Signal Objects

If a signal is defined in the `Signal Properties` dialog box and a signal object of the same name is defined by using the command line or in the `Model Explorer`, the potential exists for ambiguity when the Simulink engine attempts to resolve the symbol representing the signal name. One way to resolve the ambiguity is to specify that a signal resolve to a Simulink data object. Select the **Signal name must resolve to Simulink signal object** option in the `Signal Properties` dialog box. You cannot specify the **Storage class** property on the **Code Generation** tab in the `Signal Properties` dialog box.



As the preceding figure shows, the **Storage class** menu is disabled because it is up to the `SinSig Simulink.Signal` object to specify its own storage class.

The signal and signal objects `SinSig` both have `SimulinkGlobal` storage class. Therefore, `SinSig` resolves to the signal object `SinSig`.

Note The rules for compatibility between block states/signal objects are identical to those given for signals/signal objects.

Related Examples

- “Access Signal, State, and Parameter Data During Execution” on page 14-3
- “Design Data Interface by Configuring Inport and Outport Blocks” on page 14-131
- “Maximize Signal Storage Optimization” on page 14-144

- “Group Signals into Structures in the Generated Code Using Buses” on page 14-136
- “Control Signal Data Types”
- “Signal Representation in Generated Code” on page 14-109
- “Storage Classes for Signals Used with Model Blocks” on page 8-22
- “Control Signal and State Initialization in the Generated Code” on page 14-145
- “Default Data Structures in the Generated Code” on page 14-16
- “Virtualized Output Ports Optimization”

Design Data Interface by Configuring Inport and Outport Blocks

The data interface of a model is the means by which the model exchanges data (for example, signal values) with other, external models or systems. Customize the data interface of a model to:

- Enable integration of the generated code with your own code.
- Improve traceability and readability of the code.

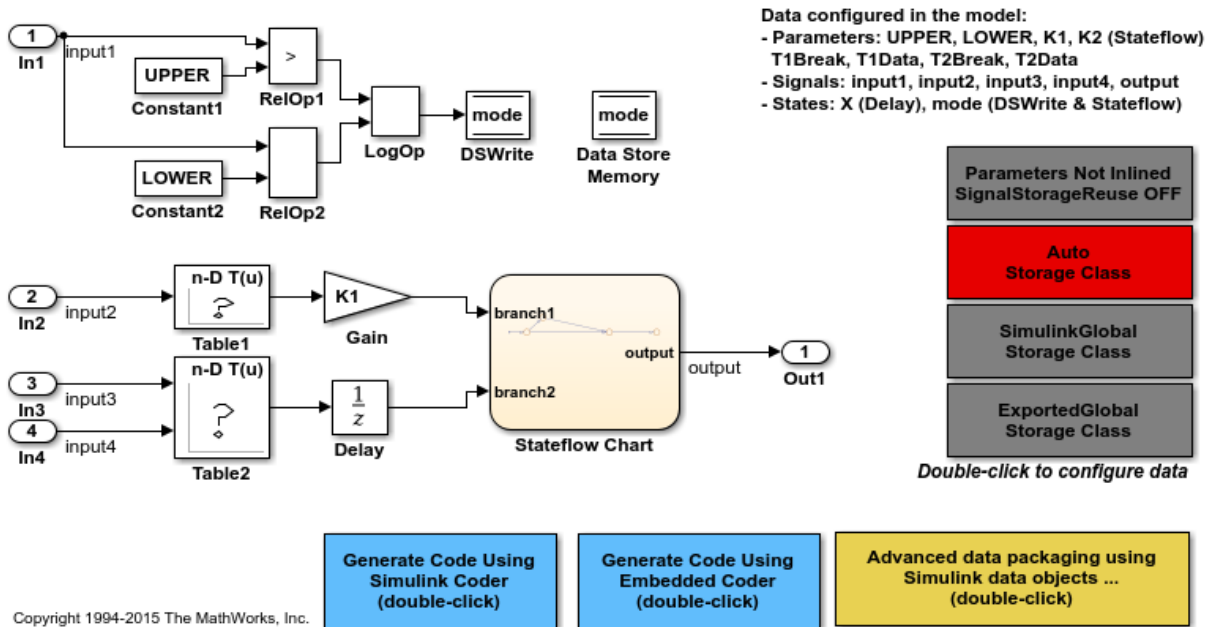
At the top level of a model, Inport and Outport blocks represent the input and output signals of the model. To customize the data interface in the generated code, configure these blocks. Early in the design process, when a model can contain unconnected Inport and Outport blocks, use this technique to specify the interface before developing the internal algorithm.

When you apply storage classes to Inport and Outport blocks, each block appears in the generated code as a field of a global structure or as a separate global variable that the generated algorithm references directly. If you have Embedded Coder, you can use function prototype control instead of storage classes to pass data into and out of the model `step` function as formal parameters. See “Control Generation of Function Prototypes”.

Design Data Interface

Open the example model `rtwdemo_basicsc`.

```
open_system('rtwdemo_basicsc')
```



Copyright 1994-2015 The MathWorks, Inc.

In the model, select **View > Model Data**.

In the Model Data Editor, select the **Inports/Outports** tab. Each row in the table represents an Inport or Outport block.

Name the output signal of the Outport block labeled Out1. Set **Signal Name** to output_sig.

For each of the Inport blocks, set **Data Type** to single or to a different data type. Due to the data type inheritance settings that the other blocks in the model use by default, downstream signals use the same or a similar data type.

Optionally, configure other design attributes such as **Min** and **Max** (minimum and maximum values).

Set the **Change View** drop-down list to Code.

For the Outport block and all of the Inport blocks, set **Storage Class** to ExportedGlobal. To configure all of the blocks at once, select all of the rows in the table.

To configure the blocks and signals, you can use these commands at the command prompt.

```
temp = Simulink.Signal;
temp.StorageClass = 'ExportedGlobal';

portHandles = get_param('rtwdemo_basicsc/In1','portHandles');
outPortHandle = portHandles.Outport;
set_param(outPortHandle,'SignalObject',temp);

portHandles = get_param('rtwdemo_basicsc/In2','portHandles');
outPortHandle = portHandles.Outport;
set_param(outPortHandle,'SignalObject',temp);

portHandles = get_param('rtwdemo_basicsc/In3','portHandles');
outPortHandle = portHandles.Outport;
set_param(outPortHandle,'SignalObject',temp);

portHandles = get_param('rtwdemo_basicsc/In4','portHandles');
outPortHandle = portHandles.Outport;
set_param(outPortHandle,'SignalObject',temp);

set_param('rtwdemo_basicsc/Out1','SignalName','output_sig',...
    'SignalObject',temp)
```

Generate code from the model.

```
rtwbuild('rtwdemo_basicsc');

### Starting build procedure for model: rtwdemo_basicsc
### Successful completion of build procedure for model: rtwdemo_basicsc
```

View the generated file `rtwdemo_basicsc.c`. Because you applied the storage class `ExportedGlobal` to the Inport and Outport blocks, the code creates separate global variables that represent the inputs and the output.

```
file = fullfile('rtwdemo_basicsc_grt_rtw','rtwdemo_basicsc.c');
rtwdemodbtype(file,'/* Exported block signals */','real32_T output_sig;',1,1)

/* Exported block signals */
real32_T input1;          /* '<Root>/In1' */
real32_T input2;          /* '<Root>/In2' */
real32_T input3;          /* '<Root>/In3' */
real32_T input4;          /* '<Root>/In4' */
```

```
real32_T output_sig;          /* '<Root>/Out1' */
```

The generated algorithm in the model `step` function directly references these global variables to calculate and store the output signal value, `output_sig`.

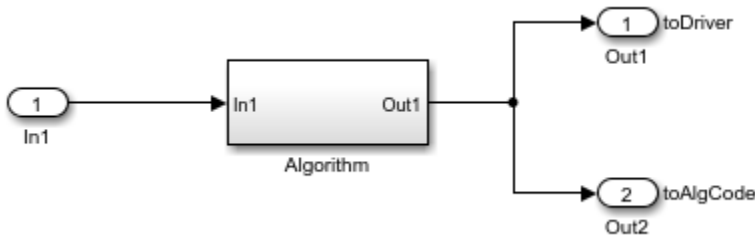
While you use the Model Data Editor to configure the interface of a system, consider using the interface display to view the system inputs and outputs (Inport and Outport blocks) at a high level. See “Configure Data Interface for Component”.

Route Signal Data to Multiple Outputs

You can route a single signal to multiple Outport blocks and apply a different storage class to each Outport. For example, use this technique to send signal data to a custom function (such as a device driver) and to a global variable that your custom algorithmic code can use:

- 1 Branch the target signal line to each Outport block.
- 2 For more efficient code, set the storage class of the target signal line to `Auto` (the default). Optimizations can then eliminate the signal line from the generated code.
- 3 Use the Model Data Editor to apply the custom storage class `GetSet` to one Outport block and `ExportToFile` to the other Outport block. Apply a signal name to each block.

```
open_system('ex_route_sig')
```



Limitations

You cannot apply a storage class to an Outport block if the input to the block has a variable size. Instead, apply the storage class to the signal line.

Related Examples

- “Analyze the Generated Code Interface”

- “Trace Connections Using Interface Display”
- “Interface Design”
- “Signal Representation in Generated Code” on page 14-109
- “Configure Generated Code According to Interface Control Document”
- “Control Generation of Function Prototypes”
- “Configure Data Properties by Using a Table”

Group Signals into Structures in the Generated Code Using Buses

Buses in a model represent multiple signals as a single signal line. You can use nonvirtual buses to create signal structures in the generated code.

You can generate flat and nested structures. See “Structures of Signals”.

For basic information about buses in models, see “Buses”.

Import or Export Structure Variable and Definition

This example shows how to export the definition of a bus type and the declaration of a nonvirtual bus signal in generated code. You can control the bus type definition and the bus signal declaration in the generated code independently of each other.

Explore Example Model

- 1 Open the model `rtwdemo_slbus`.

The model creates a `Simulink.Bus` object `BusObject` in the base workspace.

- 2 Update the model diagram to display thick lines for composite signals, including buses.

The bus signal `S1` uses the bus type that `BusObject` defines.

Control Scope of Bus Type

To control code generated for a bus type that a `Simulink.Bus` object defines, adjust the code generation settings for the object.

The generated code represents nonvirtual bus types with `struct` definitions. The `struct` definitions contain fields corresponding to the elements of the bus.

- 1 At the command prompt, specify the `DataScope` property of `BusObject` as `'Exported'`.

```
BusObject.DataScope = 'Exported';
```

- 2 Specify the `HeaderFile` property of the object as `'myBusTypeHdr.h'`.

```
BusObject.HeaderFile = 'myBusTypeHdr.h';
```


Control Scope of Bus Signal

You can use a `Simulink.Signal` object to control code generated for signals in a model, including composite signals such as buses. To control code generated for a nonvirtual bus signal, apply a storage class to the bus.

- 1 View the model in the Simulink Editor.
- 2 In the signal properties dialog box for the bus signal `S1`, select the option **Signal name must resolve to Simulink.Signal object**.
- 3 At the command prompt, create a `Simulink.Signal` object to represent the bus signal `S1`. Specify the data type of the object as `BusObject`.

```
S1 = Simulink.Signal;  
S1.DataType = 'Bus: BusObject';
```

- 4 Specify the storage class of the object as `ExportedGlobal`.

```
S1.StorageClass = 'ExportedGlobal';
```

For more information about applying storage classes to signals, see “Control Signals and States in Code by Applying Storage Classes” on page 14-120.

Generate and Inspect Code

- 1 In the Simulink Editor, double-click the blue box labeled **Generate Code Using Simulink Coder**.
- 2 In the code generation report, view the generated file `myBusTypeHdr.h`. The code in the header file represents the bus type `BusObject` with a `struct` definition.

```
typedef struct {  
    real_T temperature;  
    real_T heat;  
    real_T pressure[20];  
} BusObject;
```

- 3 View the file `rtwdemo_slbus.c`. The code exports the declaration of the signal `S1` and uses the structure type `BusObject` in the declaration.

```
/* Exported block signals */  
BusObject S1;
```

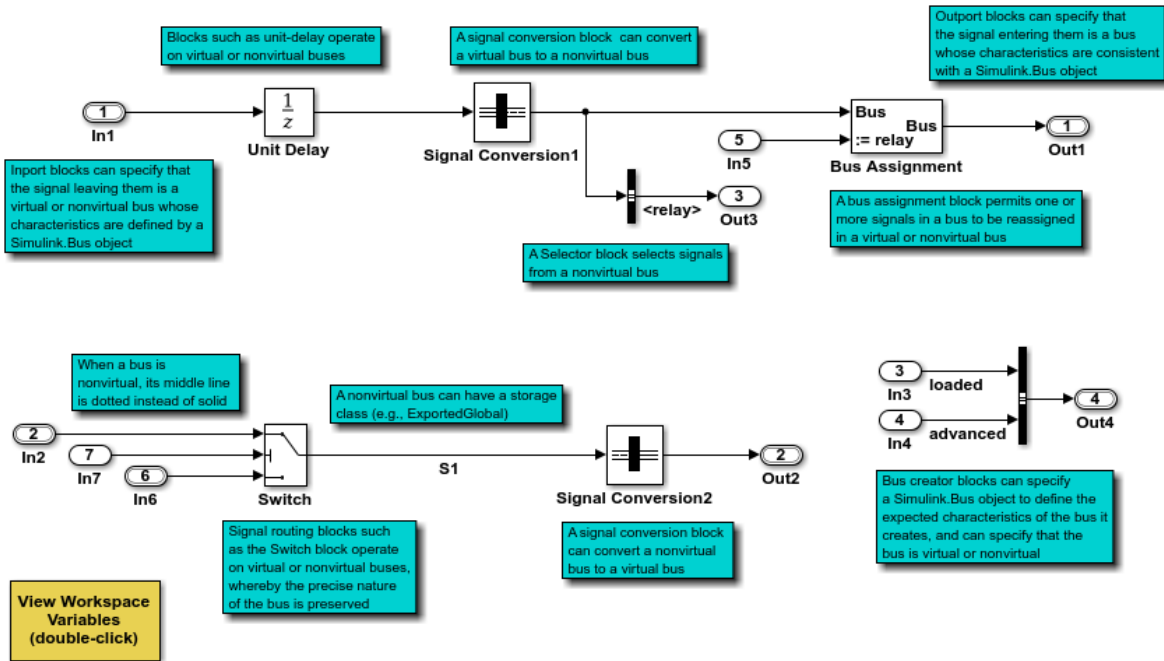
Model Data Specifications with Bus Signals

Simulink® buses and bus objects are powerful modeling constructs that provide excellent data specification capabilities, particularly to specify system interfaces.

Open Example Model

Open the example model `rtwdemo_slbus`.

```
open_system('rtwdemo_slbus');
```



Simulink buses and bus objects are powerful modeling constructs that provide excellent data specification capabilities, particularly to specify system interfaces.

To view the signal attributes for this model, choose Simulation > Update Diagram from the model toolbar, or press Ctrl+D. To view the Simulink bus objects in the base workspace, double-click the yellow button above.

Generate Code Using
Simulink Coder
(double-click)

Generate Code Using
Embedded Coder
(double-click)

To view the signal attributes for this model, on the model toolbar, choose **Simulation > Update Diagram**, or press **Ctrl+D**. To view the Simulink bus objects in the base workspace, double-click the yellow button in the model.

Arrays of Structures

You can further package multiple consistent bus signals into an array of buses. The array of buses appears in the generated code as an array of structures. To create arrays of buses, see “Combine Buses into an Array of Buses”.

Structure Padding

By default, the code generator does not explicitly add padding fields to structure types. Structure types can appear in the generated code through, for example, the default data structures (see “Default Data Structures in the Generated Code” on page 14-16), `Simulink.Bus` objects, and parameter structures that you use in a model.

However, when you use a code replacement library with Embedded Coder, you can specify data alignment (including structure padding) as part of the replacement library. For more information, see “Provide Data Alignment Specifications for Compilers”.

See Also

`Simulink.Bus` | `Simulink.Signal`

Related Examples

- “Generate Efficient Code for Bus Signals” on page 14-140
- “Specify Sample Times for Signal Elements”
- “Signal Representation in Generated Code” on page 14-109
- “Control Signals and States in Code by Applying Storage Classes” on page 14-120
- “Buses”

Generate Efficient Code for Bus Signals

In this section...
“Code Efficiency for Bus Signals” on page 14-140
“Set Bus Diagnostics” on page 14-141
“Optimize Virtual and Nonvirtual Buses” on page 14-141

In a model, you use bus signals to package multiple signals together into a single signal line. You can create virtual or nonvirtual bus signals. The representation in the generated code depends on:

- For a virtual bus, the generated code appears as if the bus did not exist.
- Generated code for a nonvirtual bus represents the bus data with a structure. When you want to trace the correspondence between the model and the code, the use of a structure in the generated code can be helpful. To generate structures using nonvirtual bus signals, see “Group Signals into Structures in the Generated Code Using Buses” on page 14-136.

For general information about buses, see and “Virtual and Nonvirtual Buses”.

To generate efficient code from models that contain bus signals, eliminate unnecessary data copies by following best practices as you construct the model.

Code Efficiency for Bus Signals

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

- Setting bus diagnostic configuration parameters can make model development easier.
- The bus implementation techniques, and the choice of a nonvirtual or virtual bus, can influence the speed, size, and clarity of the generated code.
- Some useful bus implementation techniques are not immediately obvious.

When you work with buses, these guidelines help you to improve the results. The guidelines describe techniques to:

- Simplify the layout of the model.
- Increase the efficiency of generated code.
- Define data structures for function (subsystem) interfaces.

- Define data structures that match existing data structures in external C code.

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

Set Bus Diagnostics

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

The screenshot shows the 'Buses' configuration pane with the following settings:

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

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

Optimize Virtual and Nonvirtual Buses

Virtual buses are graphical conveniences that do not affect generated code. As a result, the code generation engine is able to fully optimize the signals in the bus. Use virtual buses rather than nonvirtual buses wherever possible. You can convert between virtual and nonvirtual buses by using **Signal Conversion** blocks. In some cases, Simulink automatically converts a virtual bus to a nonvirtual bus when required. For example, a Stateflow chart converts an input virtual bus to a nonvirtual bus.

To bundle function-call signals, you must use a virtual bus.

You must use nonvirtual buses for:

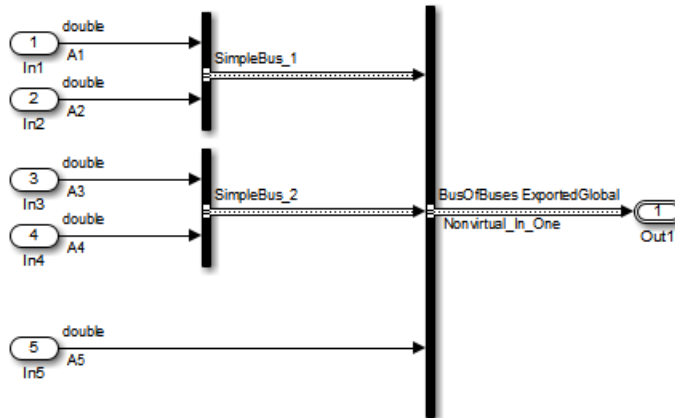
- Nonauto storage classes
- Generating a specific structure from the bus
- Root-level Inport or Outport blocks when the bus has mixed data types

Avoid Nonlocal Nested Buses in Nonvirtual Buses

Buses can contain subordinate buses. To generate efficient code, set the storage classes of subordinate buses to **Auto**. Setting the storage class to **Auto** eliminates:

- Allocation of redundant memory for the subordinate bus signal and for the parent bus signal
- Additional copy operations (copying data to the subordinate bus, and then copying from the subordinate bus to the final bus)

This model contains nonvirtual bus signals. The subordinate bus signals **Sub_Bus_1** and **Sub_Bus_2** use the storage class **Auto**.



The generated code algorithm efficiently assigns the input signal data to the bus signals.

```
void ex_nonvirtual_buses_step(void)
{
  Nonvirtual_In_One.SimpleBus_1.A1 = A1;
  Nonvirtual_In_One.SimpleBus_1.A2 = A2;
  Nonvirtual_In_One.SimpleBus_2.A3 = A3;
  Nonvirtual_In_One.SimpleBus_2.A4 = A4;
  Nonvirtual_In_One.A5 = A5;
}
```

```
}
```

See Also

`Simulink.Bus`

Related Examples

- “Group Signals into Structures in the Generated Code Using Buses” on page 14-136
- “Specify Sample Times for Signal Elements”

Maximize Signal Storage Optimization

The value of the “Maximum stack size (bytes)” parameter, on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box constrains the use of stack space used by local block output variables. The command-line equivalent for this parameter is `MaxStackSize`. If the accumulated size of variables in local memory exceeds `MaxStackSize`, the product places subsequent local variables in global memory space.

If it is important that you maximize potential for signal storage optimization, then set `MaxStackSize` to accommodate the size and number of signals in your model. This minimizes overflow into global memory space and maximizes use of local memory. Local variables offer more optimization potential through mechanisms such as expression folding and buffer reuse. See “Customize Stack Space Allocation” on page 31-93 for more information.

Related Examples

- “Customize Stack Space Allocation” on page 31-93
- “Signal Representation in Generated Code” on page 14-109

Control Signal and State Initialization in the Generated Code

To initialize signals and discrete states with custom values for simulation and code generation, you can use signal objects and block parameters. Data initialization increases application reliability and is a requirement of safety critical applications. Initializing signals for both simulation and code generation can expedite transitions between phases of Model-Based Design.

For basic information about specifying initial values for signals and discrete states in a model, see “Initialize Signals and Discrete States”.

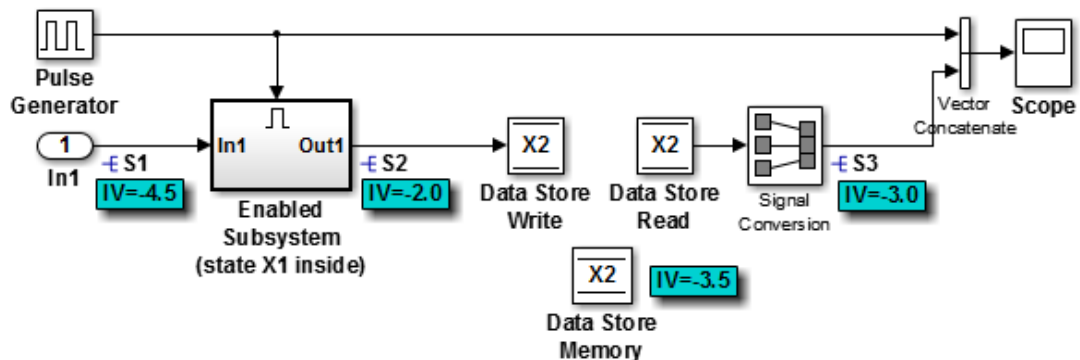
Signal and State Initialization in the Generated Code

The initialization behavior for code generation is the same as that for model simulation with the following exceptions:

- RSim executables can use the **Data Import/Export** pane of the Configuration Parameters dialog box to load input values from MAT-files. GRT and ERT executables cannot load input values from MAT-files.
- The initial value for a block output signal or root level input or output signal can be overwritten by an external (calling) program.
- Setting the initial value for persistent signals is relevant if the value is used or viewed by an external application.

When you generate code, initialization statements are placed in *model.c* or *model.cpp* in the model's initialize code.

For example, consider the model *rtwdemo_sigobj_iv*.



If you create and initialize signal objects in the base workspace, the code generator places initialization code for the signals in the file `rtwdemo_sigobj_iv.c` under the `rtwdemo_sigobj_iv_initialize` function, as shown below.

```
/* Model initialize function */
void rtwdemo_sigobj_iv_initialize(void)
{
    .
    .
    .
    /* exported global signals */
    S3 = -3.0;
    S2 = -2.0;
    .
    .
    .
    /* exported global states */
    X1 = 0.0;
    X2 = 0.0;
    /* external inputs */
    S1 = -4.5;
    .
    .
    .
}
```

The following code shows the initialization code for the enabled subsystem's Unit Delay block state `X1` and output signal `S2`.

```
void MdlStart(void) {
    .
    .
    .
    /* InitializeConditions for UnitDelay: '<S2>/Unit Delay' */
    X1 = aa1;
    /* Start for enable system: '<Root>/Enabled Subsystem (state X1 inside)' */
    /* virtual outports code */
    /* (Virtual) Outport Block: '<S2>/Out1' */
    S2 = aa2;
}
}
```

For an enabled subsystem, the initial value is also used as a reset value if the subsystem's Outport block parameter **Output when disabled** is set to **reset**. The following code from `rtwdemo_sigobj_iv.c` shows the assignment statement for `S3` as it appears in the model output function `rtwdemo_sigobj_iv_output`.

```
/* Model output function */
static void rtwdemo_sigobj_iv_output(void)
```

```
{
    :
    :
    /* Disable for enable system: '<Root>/Enabled Subsystem (state X1 inside)' */
    /* (Virtual) Output Block: '<S2>/Out1' */
    S2 = aa2;
```

Generate Tunable Initial Conditions

You can represent initial conditions for signals and states by creating tunable global variables in the generated code. These variables allow you to restart an application by using initial conditions that are stored in memory.

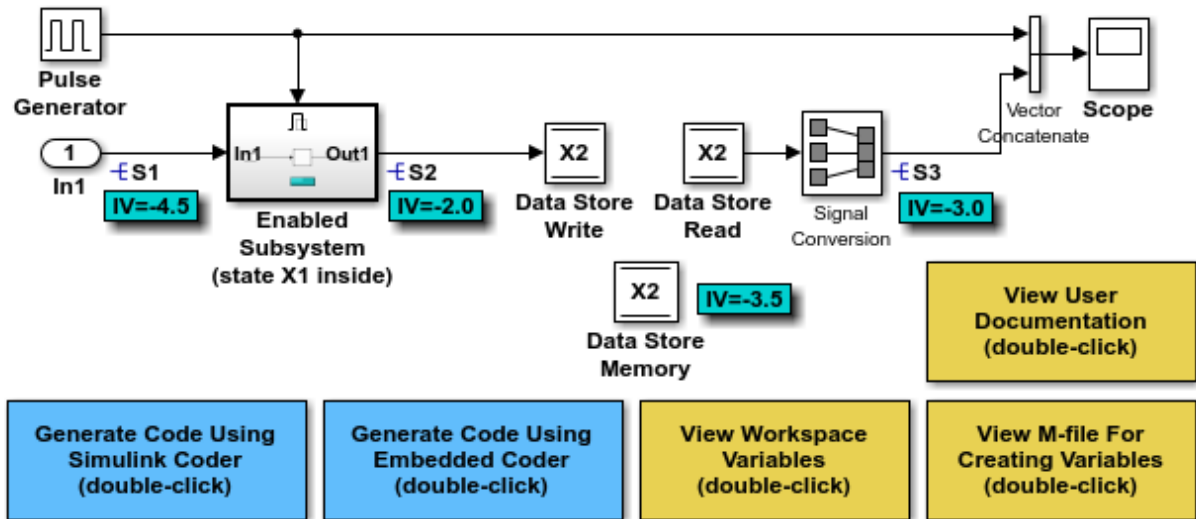
If you set **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** to **Tunable**, initial conditions appear as tunable fields of the global parameters structure.

Whether you set **Default parameter behavior** to **Tunable** or **Inlined**, you can use a tunable parameter to specify the **InitialValue** property of a signal object or the **Initial condition** parameter of a block. For basic information about tunable parameters, see “Block Parameter Representation in the Generated Code”.

This example shows how to use tunable parameters to specify initial conditions for signals and states.

Explore Example Model

Open the example model `rtwdemo_sigobj_iv`. The signal `S2` uses a `Simulink.Signal` object in the base workspace.



Copyright 1994-2015 The MathWorks, Inc.

Double-click the `Simulink.Signal` object `S2` to view its properties. The **Initial value** property is set to `aa2`. The object uses the variable `aa2` to specify an initial condition for the signal `S2`. The **Storage class** property is set to `ExportedGlobal`. To use a `Simulink.Signal` object to initialize a signal, the signal object must use a storage class other than `Auto` or `SimulinkGlobal`.

On the **Optimization > Signals and Parameters** pane in the Configuration Parameters dialog box, click **Configure**. The variable `aa2` is a tunable parameter that uses the storage class `ExportedGlobal`.

In the model, open the Enabled Subsystem. In the Output port block dialog box, the parameter **Output when disabled** is set to `reset`. When the subsystem becomes disabled, the output signal `S2` resets to the initial value `aa2`.

Open the Unit Delay block dialog box. On the **State Attributes** tab, the **State name** box is set to `X1`.

Open the Enable block dialog box. The parameter **States when enabling** is set to `reset`. When the subsystem transitions from a disabled state to an enabled state, it resets internal block states, such as `X1`, to their initial values.

In the base workspace, double-click the `Simulink.Signal` object `X1` to view its properties. The **Initial value** property is set to `aa1`.

Double-click the `Simulink.Parameter` object `aa1` to view its properties. The **Storage class** property is set to `ExportedGlobal`. You can generate tunable initial conditions for block states by using tunable parameters such as `aa1` and `Simulink.Signal` objects such as `X1`.

Generate and Inspect Code

Generate code with the example model.

```
### Starting build procedure for model: rtwdemo_sigobj_iv
### Successful completion of build procedure for model: rtwdemo_sigobj_iv
```

In the code generation report, view the file `rtwdemo_sigobj_iv.c`. The code uses global variables to represent the block state `X1` and the signal `S2`.

```
/* Exported block states */
real_T X1;                                /* '<S2>/Unit Delay' */

/* Exported block signals */
real_T S1;                                /* '<Root>/In1' */
real_T S3;                                /* '<Root>/Signal Conversion' */
real_T S2;                                /* '<S2>/Unit Delay' */
```

The code uses global variables to represent the tunable parameters `aa1` and `aa2`.

```
/* Exported block parameters */
real_T aa1 = -2.5;                        /* Variable: aa1 */

real_T aa2 = -2.0;                        /* Variable: aa2 */
```

The model initialization function uses the tunable parameter `aa1` to initialize the state `X1`. The function also uses the tunable parameter `aa2` to initialize the signal `S2`.

```
/* SystemInitialize for Enabled SubSystem: '<Root>/Enabled Subsystem (state X1 inside
/* InitializeConditions for UnitDelay: '<S2>/Unit Delay' */
X1 = aa1;

/* SystemInitialize for Output: '<S2>/Out1' */
S2 = aa2;
```

In the model step function, when the Enabled Subsystem transitions from a disabled state to an enabled state, the Unit Delay block state X1 resets to its initial value.

```
if (rtb_PulseGenerator > 0) {
    if (!rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M) {
        /* InitializeConditions for UnitDelay: '<S2>/Unit Delay' */
        X1 = aa1;
        rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M = true;
    }
}
```

If the Enabled Subsystem becomes disabled during code execution, the algorithm uses the tunable initial condition **aa2** to set the value of the signal **S2**.

```
} else {

    if (rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M) {
        /* Disable for Output: '<S2>/Out1' */
        S2 = aa2;
        rtwdemo_sigobj_iv_DW.EnabledSubsystemstateX1inside_M = false;
    }
}
```

Generate Tunable Initial Condition Structure for Bus Signal

When you use a MATLAB® structure to specify initialization values for the signal elements in a bus, you can create a tunable global structure in the generated code.

If you set **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** to **Tunable**, the initial condition appears as a tunable substructure of the global parameters structure.

Whether you set **Default parameter behavior** to **Tunable** or **Inlined**, you can specify the initial condition by using a tunable `Simulink.Parameter` object whose value is a structure. If you apply a storage class other than `Auto` to the parameter object, the structure is tunable in the generated code.

To generate efficient code by avoiding data type mismatches between the structure and the bus signal, use either:

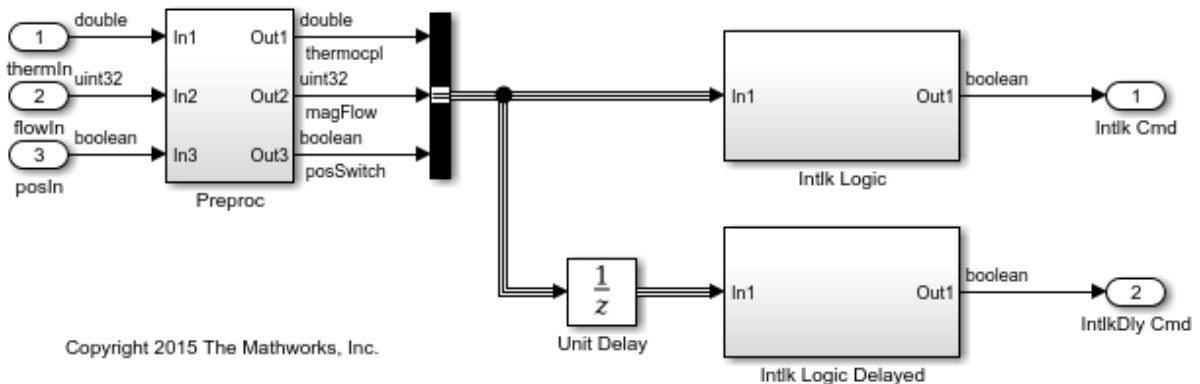
- Typed expressions to specify the values of the structure fields. Match the data type of each field with the data type of the corresponding signal element.
- A `Simulink.Bus` object to control the data types of the structure fields and the signal elements.

For basic information about using structures to initialize bus signals, and to decide how to control field data types, see “Specify Initial Conditions for Bus Signals”.

Generate Tunable Initial Condition Structure

This example shows how to use a tunable structure parameter to initialize a virtual bus signal.

Open the example model `rtwdemo_tunable_init_struct`.



In the Inport block dialog boxes, open the **Signal Attributes** tab. Each Inport uses a different output data type.

Open the Bus Creator block dialog box. The block output is a virtual bus.

In the Configuration Parameters dialog box, open the **Optimization > Signals and Parameters** pane. The configuration parameter **Default parameter behavior** is set to **Tunable**. By default, block parameters, including initial conditions, appear in the generated code as tunable fields of the global parameters structure.

Open the Unit Delay block dialog box. Set **Initial condition** to a structure that specifies an initial condition for each of the three signal elements. To generate efficient code,

match the data types of the structure fields with the data types of the corresponding signal elements.

```
set_param('rtwdemo_tunable_init_struct/Unit Delay','InitialCondition',...
'struct(''thermocpl'',15.23,'magFlow',uint32(79),'posSwitch',false)')
```

Generate code from the example model.

```
### Starting build procedure for model: rtwdemo_tunable_init_struct
### Successful completion of build procedure for model: rtwdemo_tunable_init_struct
```

In the code generation report, view the file `rtwdemo_tunable_init_struct_types.h`. The code defines a structure type whose fields use the data types that you specified in the `struct` expression.

```
#ifndef DEFINED_TYPEDEF_FOR_struct_mqGi1jsItE0G7cf1bNqMu_
#define DEFINED_TYPEDEF_FOR_struct_mqGi1jsItE0G7cf1bNqMu_

typedef struct {
    real_T thermocpl;
    uint32_T magFlow;
    boolean_T posSwitch;
} struct_mqGi1jsItE0G7cf1bNqMu;
```

View the file `rtwdemo_tunable_init_struct.h`. The `struct` type definition of the global parameters structure contains a substructure, `UnitDelay_InitialCondition`, which represents the **Initial condition** parameter of the Unit Delay block.

```
struct P_rtwdemo_tunable_init_struct_T_ {
    struct_mqGi1jsItE0G7cf1bNqMu UnitDelay_InitialCondition; /* Mask Parameter: UnitDelay_InitialCondition */
};
```

View the file `rtwdemo_tunable_init_struct_data.c`. This source file allocates memory for the global parameters structure. The substructure `UnitDelay_InitialCondition` appears.

```
/* Block parameters (auto storage) */
P_rtwdemo_tunable_init_struct_T rtwdemo_tunable_init_struct_P = {
    {
        15.23,
        79U,
        0
    }
};
```



```
}, /* Mask Parameter: UnitDelay_InitialCondition
```

View the file `rtwdemo_tunable_init_struct.c`. The model initialization function uses the fields of the substructure to initialize the block states.

```
/* InitializeConditions for UnitDelay: '<Root>/Unit Delay' */
rtwdemo_tunable_init_struct_DW.UnitDelay_1_DSTATE =
    rtwdemo_tunable_init_struct_P.UnitDelay_InitialCondition.thermocpl;

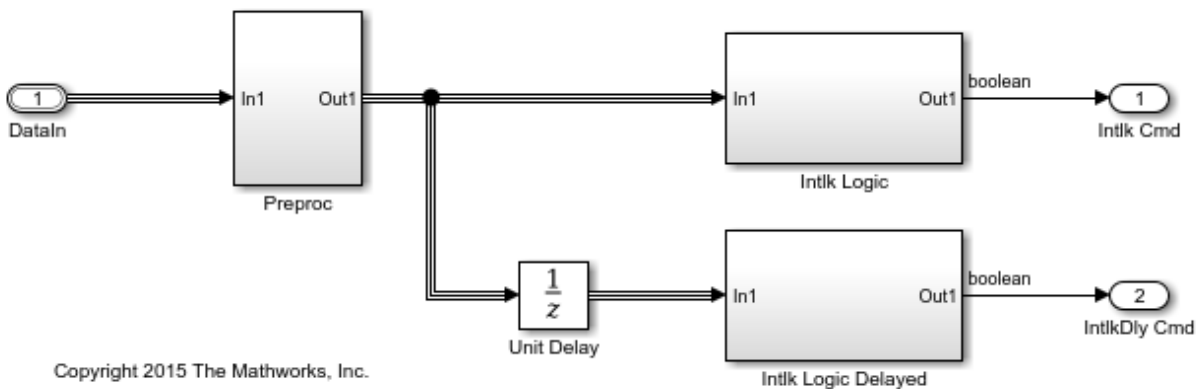
/* InitializeConditions for UnitDelay: '<Root>/Unit Delay' */
rtwdemo_tunable_init_struct_DW.UnitDelay_2_DSTATE =
    rtwdemo_tunable_init_struct_P.UnitDelay_InitialCondition.magFlow;

/* InitializeConditions for UnitDelay: '<Root>/Unit Delay' */
rtwdemo_tunable_init_struct_DW.UnitDelay_3_DSTATE =
    rtwdemo_tunable_init_struct_P.UnitDelay_InitialCondition.posSwitch;
```

Use Bus Object to Specify Data Types

If you create a bus object, you can use it to specify the data type of the bus signal and the tunable initial condition structure. Before code generation, the `Simulink.Parameter` object casts the values of the structure fields to the data types of the signal elements. For basic information about creating bus objects and using them in models, see “When to Use Bus Objects”.

Open the example model `rtwdemo_init_struct_busobj`.



In the base workspace, double-click the `Simulink.Bus` object `ComponentData`. The object defines three signal elements: `thermocpl`, `magFlow`, and `posSwitch`. The elements each use a different data type.

Open the block dialog box for the Inport block `DataIn`. The output data type is set to `Bus: ComponentData`.

Create the tunable structure parameter `initStruct`. You can specify the field values by using untyped expressions. To improve readability, specify the field `posSwitch` with a Boolean value. Specify the `DataType` property of the parameter object as `'Bus: ComponentData'`.

```
initStruct = struct(...
    'thermocpl',15.23,...
    'magFlow',79,...
    'posSwitch',false ...
);

initStruct = Simulink.Parameter(initStruct);
initStruct.StorageClass = 'ExportedGlobal';
initStruct.DataType = 'Bus: ComponentData';
```

In the Unit Delay block dialog box, specify **Initial condition** as `initStruct`.

Generate code from the example model.

```
### Starting build procedure for model: rtwdemo_init_struct_busobj
### Successful completion of build procedure for model: rtwdemo_init_struct_busobj
```

In the code generation report, view the file `rtwdemo_init_struct_busobj_types.h`. The code creates a structure type `ComponentData` whose fields use the data types in the bus object.

```
#ifndef DEFINED_TYPEDEF_FOR_ComponentData_
#define DEFINED_TYPEDEF_FOR_ComponentData_

typedef struct {
    real_T thermocpl;
    uint32_T magFlow;
    boolean_T posSwitch;
} ComponentData;
```

View the file `rtwdemo_init_struct_busobj.c`. The code creates a global variable to represent the tunable parameter object `initStruct`.

```

/* Exported block parameters */
ComponentData initStruct = {
    15.23,
    79U,
    0
};
/* Variable: initStruct

```

The model initialization function uses the structure fields to initialize the block states.

```

/* InitializeConditions for UnitDelay: '<Root>/Unit Delay' */
rtwdemo_init_struct_busobj_DW.UnitDelay_1_DSTATE = initStruct.thermocpl;

/* InitializeConditions for UnitDelay: '<Root>/Unit Delay' */
rtwdemo_init_struct_busobj_DW.UnitDelay_2_DSTATE = initStruct.magFlow;

/* InitializeConditions for UnitDelay: '<Root>/Unit Delay' */
rtwdemo_init_struct_busobj_DW.UnitDelay_3_DSTATE = initStruct.posSwitch;

```

To change the data type of any of the three signal elements, specify the new type in the bus object. The signal element in the model uses the new type. Before simulation and code generation, the parameter object `initStruct` casts the corresponding structure field to the new type.

See Also

State Reader | State Writer

Related Examples

- “Initialization Behavior Summary for Signal Objects”
- “Specify Initial Conditions for Bus Signals”
- “Control Signals and States in Code by Applying Storage Classes” on page 14-120
- “Generate Code That Responds to Initialize, Reset, and Terminate Events” on page 12-2
- “Block Parameter Representation in the Generated Code” on page 14-43
- “Signal Representation in Generated Code” on page 14-109
- “Remove Initialization Code”

Continuous Block State Naming in Generated Code

In this section...
“Default Block State Naming Convention” on page 14-156
“Define User Block State Names” on page 14-157

To determine the variable or field name generated for a continuous block state, do one of the following:

- Use a default name created by the code generator.
- In the block parameter dialog box, in the **State name** field, specify a name.

These names apply to the following continuous blocks:

- Integrator
- Integrator Limited
- Second-Order Integrator
- Second-Order Integrator Limited
- PID Controller
- PID Controller (2DOF)
- State Space
- Transfer Fcn
- Variable Time Delay
- Variable Transport Delay
- Zero-Pole

To name states for other types of blocks, see “Discrete Block State Naming in Generated Code” on page 14-158.

Default Block State Naming Convention

If you do not define a symbolic name for a block state, the code generator uses the following default naming convention:

Name#_CSTATE

- **Name** is the name of the block type, such as `Integrator`. If you edit the block type name displayed in the model, the code generator uses that name in the identifier.
- **#** is a unique ID number (`#`) assigned by the code generator if multiple instances of the same block type appear in the model. The ID number is appended to **Name**.
- `_CSTATE` is a suffix that is appended to the name and ID number.

For example, if you do not specify a state name for an Integrator block, the code generator creates the identifier:

```
rtX.Integrator_CSTATE = rtP.Integrator_IC;
```

If you specify, in the **State Name** field, 'myintegratorblockname' for an Integrator block, the code generator creates this identifier:

```
rtX.myintegratorblockname = rtP.Integrator_IC;
```

Define User Block State Names

In the block parameters dialog box, you can define your own symbolic name for a block state.

- 1 In your model, double-click the block.
- 2 At the bottom of the block parameters dialog box, in the **State Name** field, enter the name that you want to use for the state.
- 3 Click **OK** or **Apply** to accept the value.
- 4 Generate code and examine it to see the new identifier name.

Related Examples

- “Control Signals and States in Code by Applying Storage Classes” on page 14-120
- “Discrete Block State Naming in Generated Code” on page 14-158

Discrete Block State Naming in Generated Code

In this section...
“Default Block State Naming Convention” on page 14-159
“Define User Block State Names” on page 14-160

To determine the variable or field name generated for a block's state, you can:

- Use a default name generated by the code generator
- Define a symbolic name by using the **State name** field of the **State Attributes** tab in a block dialog box

To name states for certain continuous blocks, see “Continuous Block State Naming in Generated Code” on page 14-156.

For certain discrete block types, you can control how block states in your model are stored and represented in the generated code. In a block dialog box, using the **State Attributes** tab, you can:

- Assign symbolic names to block states in generated code.
- Control whether states declared in generated code are interfaceable (visible) to externally written code.
- Specify that states be stored in locations declared by externally written code.

Discrete blocks that you can control how block states are stored and represented in the generated code are:

- Discrete Filter
- Discrete PID Controller
- Discrete PID Controller (2DOF)
- Discrete State-Space
- Discrete-Time Integrator
- Discrete Transfer Fcn
- Discrete Zero-Pole
- Memory
- Unit Delay

- Delay

These blocks require persistent memory to store values representing the state of the block between consecutive time intervals. By default, such values are stored in a *data type work vector*. This vector is usually referred to as the *DWork* vector. It is represented in generated code as *model_DWork*, a global data structure.

If you want to interface a block state to your handwritten code, you can specify that the state is stored to a location other than the *DWork* vector. You do this by assigning a storage class to the block state.

Default Block State Naming Convention

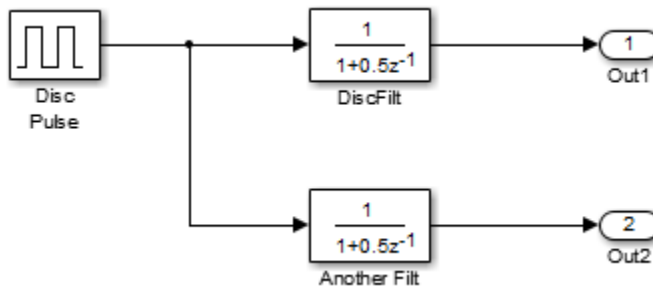
If you do not define a symbolic name for a block state, the code generator uses the following default naming convention:

`BlockType#_DSTATE`

where

- **BlockType** is the name of the block type (for example, `Discrete_Filter`). If you edit the block type name displayed in the model, the code generator uses that name in the identifier.
- **#** is a unique ID number (#) assigned by the code generator if multiple instances of the same block type appear in the model. The ID number is appended to **BlockType**.
- **_DSTATE** is a suffix that is appended to the block type and ID number.

For example, consider the model shown in the next figure.



Model with Two Discrete Filter Block States

Examine the code generated for the states of the two **Discrete Filter** blocks. Assume that:

- Neither blocks state has a user-defined name.
- The upper Discrete Filter block has **Auto** storage class (and is therefore stored in the **DWork** vector).
- The lower Discrete Filter block has **ExportedGlobal** storage class.

The states of the two Discrete Filter blocks are stored in **DWork** vectors, initialized as shown in the code:

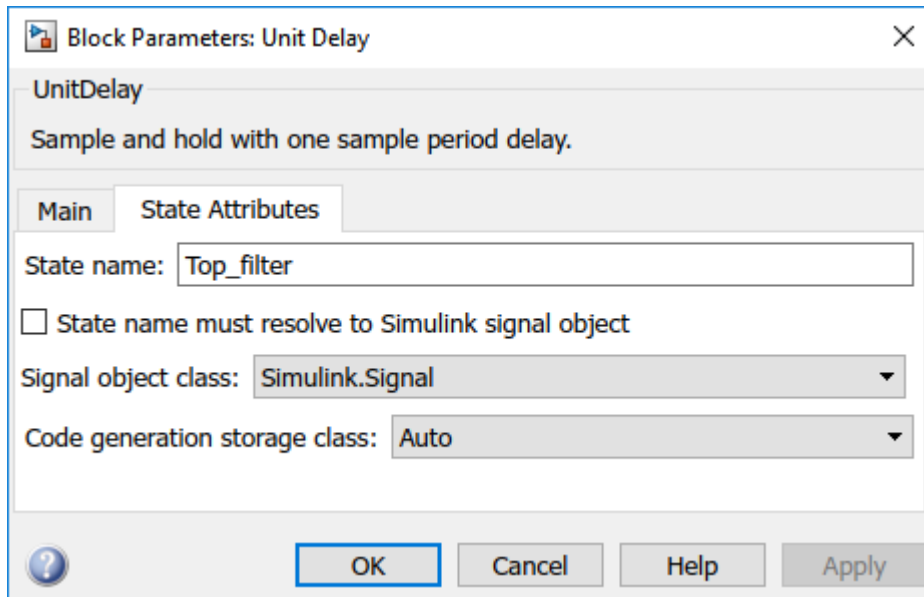
```
/* data type work */
disc_filt_states_M->Work.dwork = ((void *)
&disc_filt_states_DWork);
(void)memset((char_T *) &disc_filt_states_DWork, 0,
sizeof(D_Work_disc_filt_states));
{
    int_T i;
    real_T *dwork_ptr = (real_T *)
&disc_filt_states_DWork.DiscFilt_DSTATE;

    for (i = 0; i < 2; i++) {
        dwork_ptr[i] = 0.0;
    }
}
```

Define User Block State Names

Using the **State Attributes** tab of a block dialog box, you can define your own symbolic name for a block state:

- 1 In your block diagram, double-click the desired block. This action opens the block dialog box, containing two or more tab, which includes **State Attributes**.
- 2 Click the **State Attributes** tab.
- 3 Enter the symbolic name in the **State name** field. For example, enter the state name `Top_filter`.
- 4 Click **Apply**.



5 Click **OK**.

The following state initialization code was generated from the example model shown in “Default Block State Naming Convention” on page 14-159, under the following conditions:

- The upper Discrete Filter block has the state name `Top_filter`, and `Auto` storage class (and is therefore stored in the `DWork` vector).
- The lower Discrete Filter block has the state name `Lower_filter`, and storage class `ExportedGlobal`.

`Top_filter` is placed in the `Dwork` vector.

```
/* data type work */
disc_filt_states_M->Work.dwork = ((void *)
&disc_filt_states_DWork);
(void)memset((char_T *) &disc_filt_states_DWork, 0,
sizeof(D_Work_disc_filt_states));
disc_filt_states_DWork.Top_filter = 0.0;

/* exported global states */
```

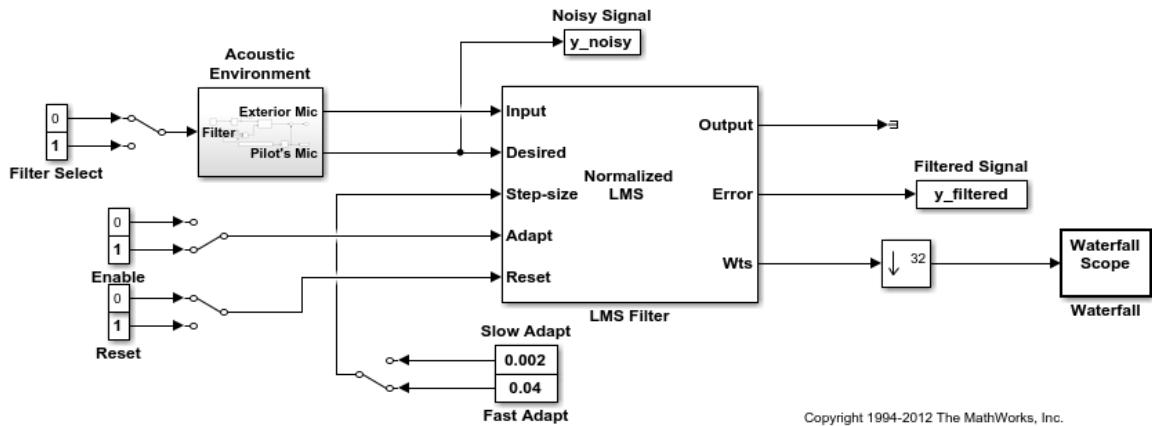
```
Lower_filter = 0.0;
```

Related Examples

- “Control Signals and States in Code by Applying Storage Classes” on page 14-120
- “Continuous Block State Naming in Generated Code” on page 14-156

Signal Processing with Fixed-Point Data

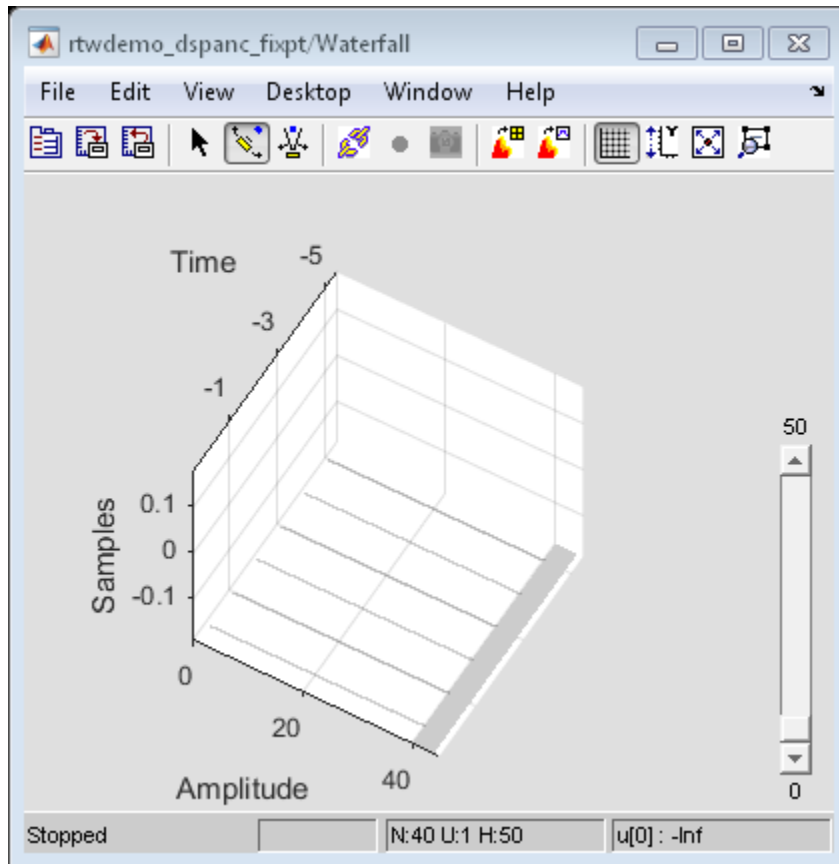
This model shows a fixed-point version of an acoustic noise canceller.



Generate Code Using
Simulink Coder
(double-click)

Generate Code Using
Embedded Coder
(double-click)

Acoustic Noise Canceler (Fixed-Point Version). See the Signal Processing Blockset documentation for details on the dspanc example.



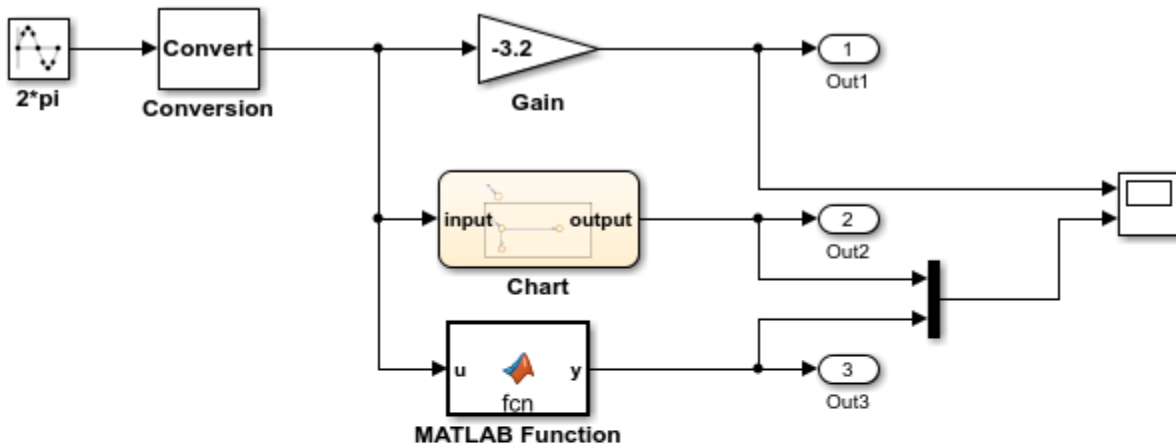
More About

- “Fixed Point”
- “Acoustic Noise Cancellation (LMS)”

Optimize Generated Code Using Fixed-Point Data with Simulink®, Stateflow®, and MATLAB®

This model shows fixed-point code generation in Simulink®, Stateflow®, and MATLAB®.

```
open_system ('rtwdemo_fixpt1');
```



This introductory model shows fixed-point code generation in Simulink, Stateflow, and MATLAB. To generate and inspect the code, double-click the blue buttons below. An HTML report detailing the code is displayed automatically.

**Generate Code Using
Simulink Coder
(double-click)**

**Generate Code Using
Embedded Coder
(double-click)**

Copyright 1994-2012 The MathWorks, Inc.

More About

- “Fixed Point”

Declare Workspace Variables as Tunable Parameters Using the Model Parameter Configuration Dialog Box

You can use the Model Parameter Configuration dialog box to declare numeric MATLAB variables in the base workspace as tunable parameters. You can select code generation options, such as storage class, for each tunable parameter.

However, it is a best practice to instead use parameter objects to declare tunable parameters. Do not use the Model Parameter Configuration dialog box to select parameter objects in the base workspace. To use parameter objects, instead of the Model Parameter Configuration dialog box, to declare tunable parameters, see “Override Default Parameter Behavior by Creating Global Variables in the Generated Code” on page 14-45.

Note You cannot use the Model Parameter Configuration dialog box to declare tunable parameters for a referenced model. Use `Simulink.Parameter` objects instead.

Declare Existing Workspace Variables as Tunable Parameters

Use the Model Parameter Configuration dialog box to declare existing workspace variables as tunable parameters for a model.

- 1 In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, click **Configure**.
- 2 In the Model Parameter Configuration dialog box, under **Source list**, select a method to populate the list of available workspace variables.
 - Select **MATLAB workspace** to view all of the numeric variables that are defined in the base workspace.
 - Select **Referenced workspace variables** to view only the numeric variables in the base workspace that the model uses. Selecting this option begins a diagram update and a search for used variables, which can take time for a large model.
- 3 In the Model Parameter Configuration dialog box, under **Source list**, select one or more workspace variables.
- 4 Click **Add to table**. The variables appear as tunable parameters under **Global (tunable) parameters**, and appear in italic font under **Source list**.

- 5 Optionally, select a parameter under **Global (tunable) parameters**, and adjust the code generation settings for the parameter. For more information about adjusting the code generation options for tunable parameters, see “Set Tunable Parameter Code Generation Options” on page 14-167
- 6 Click **OK** to apply your selection of tunable parameters and close the dialog box.

Declare New Tunable Parameters

Use the Model Parameter Configuration dialog box to declare new tunable parameters. You can use this technique to declare the names of tunable parameters, and to adjust their code generation settings, before you create the corresponding workspace variables.

- 1 In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, click **Configure**.
- 2 In the Model Parameter Configuration dialog box, under **Global (tunable) parameters**, click **New**.
- 3 Under the **Name** column, specify a name for the new tunable parameter.
- 4 Optionally, adjust the code generation settings for the new parameter. For more information about adjusting the code generation options for tunable parameters, see “Set Tunable Parameter Code Generation Options” on page 14-167
- 5 Click **OK** to apply your changes and close the dialog box.

Set Tunable Parameter Code Generation Options

To set the properties of tunable parameters listed under **Global (tunable) parameters** in the Model Parameter Configuration dialog box, select a parameter and specify a storage class and, optionally, a storage type qualifier.

Property	Description
Storage class	<p>Select one of the following to use for code generation:</p> <ul style="list-style-type: none"> • SimulinkGlobal (Auto) • ExportedGlobal • ImportedExtern • ImportedExternPointer <p>For more information about tunable parameter storage classes, see “Override Default Parameter Behavior by</p>

Property	Description
	Creating Global Variables in the Generated Code” on page 14-45.
Storage type qualifier	For variables with a storage class other than <code>SimulinkGlobal (Auto)</code> , you can add a qualifier (such as <code>const</code> or <code>volatile</code>) to the generated storage declaration. To do so, you can select a predefined qualifier from the list, or add qualifiers not in the list by typing them in. The code generator does not check the storage type qualifier for validity, and includes the qualifier text in the generated code without checking syntax .

Programmatically Declare Workspace Variables as Tunable Parameters

Tune Parameters from the Command Line

When parameters are MATLAB workspace variables, the Model Parameter Configuration dialog box is the recommended way to see or set the properties of tunable parameters. In addition to that dialog box, you can also use MATLAB `get_param` and `set_param` commands.

Note You can also use `Simulink.Parameter` objects for tunable parameters. See “Block Parameter Representation in the Generated Code” on page 14-43 for details.

The following commands return the tunable parameters and corresponding properties:

- `get_param(gcs, 'TunableVars')`
- `get_param(gcs, 'TunableVarsStorageClass')`
- `get_param(gcs, 'TunableVarsTypeQualifier')`

The following commands declare tunable parameters or set corresponding properties:

- `set_param(gcs, 'TunableVars', str)`

The argument `str` (character vector) is a comma-separated list of variable names.

- `set_param(gcs, 'TunableVarsStorageClass', str)`

The argument `str` (character vector) is a comma-separated list of storage class settings.

The valid storage class settings are

- `Auto`
- `ExportedGlobal`
- `ImportedExtern`
- `ImportedExternPointer`
- `set_param(gcs, 'TunableVarsTypeQualifier', str)`

The argument `str` (character vector) is a comma-separated list of storage type qualifiers.

The following example declares the variable `k1` to be tunable, with storage class `ExportedGlobal` and type qualifier `const`. The number of variables and number of specified storage class settings must match. If you specify multiple variables and storage class settings, separate them with a comma.

```
set_param(gcs, 'TunableVars', 'k1')
set_param(gcs, 'TunableVarsStorageClass', 'ExportedGlobal')
set_param(gcs, 'TunableVarsTypeQualifier', 'const')
```


Entry-Point Functions and Scheduling in Simulink Coder

- “Entry-Point Functions and Scheduling” on page 15-2
- “Generate Reentrant Code from Top-Level Models” on page 15-4
- “Generate C++ Class Interface to Model or Subsystem Code” on page 15-6
- “Execution of Code Generated from a Model” on page 15-9
- “Rapid Prototyping Model Functions” on page 15-21

Entry-Point Functions and Scheduling

The following functions represent entry points in the generated code for a model.

Function	Description
<code>model_initialize</code>	Initialization entry point in generated code for Simulink model. The <code>model_initialize</code> function performs model initialization and should be called once before you start executing your model.
<code>model_step</code>	Step routine entry point in generated code for Simulink model. The <code>model_step</code> function contains the output and update code for blocks in your model.
<code>model_terminate</code>	Termination entry point in generated code for Simulink model. The <code>model_terminate</code> function contains model shutdown code and should be called as part of system shutdown.

The calling interface generated for each of these functions differs depending on the value of the model parameter “Code interface packaging”:

- **C++ class** (default for C++ language) — Generates a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods.
- **Nonreusable function** (default for C language) — Generates nonreusable code. Model entry-point functions pass (`void`). Model data structures are statically allocated, global, and accessed by model entry point functions directly in the model code.
- **Reusable function** — Generates reusable, multi-instance code that is reentrant, as follows:
 - For a GRT-based model, the generated `model.c` source file contains an allocation function that dynamically allocates model data for each instance of the model. For an ERT-based model, you can use the “Use dynamic memory allocation for model initialization” option to control whether an allocation function is generated.
 - The generated code passes the real-time model data structure in, by reference, as an argument to `model_step` and the other model entry point functions.
 - The real-time model data structure is exported with the `model.h` header file.

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

The model entry points are exported with *model.h*. To call the entry-point functions from hand-written code, add an `#include model.h` directive to your code.

If the model option “Single output/update function” (selected by default) is cleared, instead of *model_step*, the following functions are generated:

- *model_output*: Contains the output code for all blocks in your model.
- *model_update*: Contains the update code for all blocks in your model.

For ERT-based models, if the model option “Terminate function required” (selected by default) is cleared, *model_terminate* is not generated.

For more information, see the reference pages for the listed functions.

Note: The function reference pages document the C language default (Nonreusable function) calling interface generated for these functions.

More About

- “Generate Reentrant Code from Top-Level Models” on page 15-4
- “Generate C++ Class Interface to Model or Subsystem Code” on page 15-6
- “Execution of Code Generated from a Model” on page 15-9
- “Rapid Prototyping Model Functions” on page 15-21

Generate Reentrant Code from Top-Level Models

To generate reentrant multi-instance code from a model, select **Reusable function** code interface packaging. When you select the **Reusable function** code interface for a GRT model:

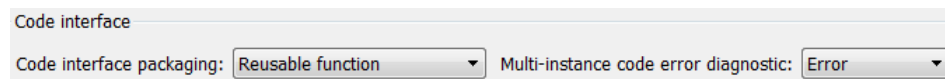
- The generated *model.c* source file contains an allocation function that dynamically allocates model data for each instance of the model.
- The generated code passes the real-time model data structure in, by reference, as an argument to *model_step* and the other model entry point functions.
- The real-time model data structure is exported with the *model.h* header file.

To configure a model to generate reusable, reentrant function code:

- 1 In the **Code Generation > Interface** pane of the Configuration Parameters dialog box, set **Code interface packaging** to the value **Reusable function**. This action enables the parameter **Multi-instance code error diagnostic**.

Note: If you have an Embedded Coder license and you have selected an ERT target for your model, selecting **Reusable function** enables additional parameters for customizing the generated reusable function interface to model code — **Pass root-level I/O as** and **Use dynamic memory allocation for model initialization**.

- 2 Examine the setting of **Multi-instance code error diagnostic**. Leave the parameter at its default value **Error** unless you have a specific need to alter the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.



- 3 Generate model code.
- 4 Examine the model entry-point function interfaces in the generated files and the HTML code generation report. For more information about generating and calling model entry-point functions, see “Entry-Point Functions and Scheduling” on page 15-2.

For an example of a model configured to generate reusable, reentrant code, open the example model *rtwdemo_reusable*. To generate GRT code for the example model, double-click the button **Generate Code Using Simulink Coder**.

More About

- “Entry-Point Functions and Scheduling” on page 15-2
- “Generate C++ Class Interface to Model or Subsystem Code” on page 15-6
- “Execution of Code Generated from a Model” on page 15-9
- “Rapid Prototyping Model Functions” on page 15-21

Generate C++ Class Interface to Model or Subsystem Code

On the Configuration Parameters dialog box, set the **Code Generation > Interface** “Code interface packaging” parameter to **C++ class** for generating a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods. The benefits of C++ class encapsulation include:

- Greater control over access to model data
- Ability to multiply instantiate model classes
- Easier integration of model code into C++ programming environments

C++ class encapsulation also works for right-click builds of nonvirtual subsystems. (For information on requirements that apply, see “Generate C++ Class Interface to Nonvirtual Subsystem Code” on page 15-7.)

Generate C++ Class Interface to Model Code

To generate encapsulated C++ class code from a GRT-based model:

- 1 Set the Configuration Parameters dialog box parameter **Code Generation > Language** to **C++**. This selection also enables C++ class code interface packaging for the model.
- 2 On the **Code Generation > Interface** pane, verify that the parameter **Code interface packaging** is set to **C++ class**.
- 3 Examine the setting of **Multi-instance code error diagnostic**. Leave the parameter at its default value **Error** unless you have a specific need to alter the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.



- 4 Generate code for the model.
- 5 Examine the C++ model class code in the generated files *model.h* and *model.cpp*. For example, the following code excerpt from the H file generated for the example model *rtwdemo_secondOrderSystem* shows the C++ class declaration for the model.

```
/* Class declaration for model rtwdemo_secondOrderSystem */
```



```

class rtwdemo_secondOrderSystemModelClass {
    /* public data and function members */
public:
    /* External outputs */
    ExtY_rtwdemo_secondOrderSystem_T rtwdemo_secondOrderSystem_Y;

    /* Model entry point functions */

    /* model initialize function */
    void initialize();

    /* model step function */
    void step();

    /* model terminate function */
    void terminate();

    /* Constructor */
    rtwdemo_secondOrderSystemModelClass();

    /* Destructor */
    ~rtwdemo_secondOrderSystemModelClass();

    /* Real-Time Model get method */
    RT_MODEL_rtwdemo_secondOrderS_T * getRTM();
    ...
};

```

For more information about generating and calling model entry-point functions, see “Entry-Point Functions and Scheduling” on page 15-2.

Note: If you have an Embedded Coder license and you have selected an ERT target for your model, you can use additional **Code Generation > Interface** pane parameters to customize the generated C++ class interface to model code.

Generate C++ Class Interface to Nonvirtual Subsystem Code

You can generate C++ class interfaces for right-click builds of nonvirtual subsystems in Simulink models, if the following requirements are met:

- The model is configured for the C++ language and C++ class code interface packaging.
- The subsystem is convertible to a Model block using the function `Simulink.SubSystem.convertToModelReference`. For referenced model conversion requirements, see the Simulink reference page `Simulink.SubSystem.convertToModelReference`.

To configure C++ class interfaces for a subsystem that meets the requirements:

- 1 Open the containing model and select the subsystem block.
- 2 Right-click the subsystem and select **C/C++ Code > Build This Subsystem**.
- 3 When the subsystem build completes, examine the C++ class interfaces in the generated files and the HTML code generation report. For more information about generating and calling model entry-point methods, see “Entry-Point Functions and Scheduling” on page 15-2.

Note: If you have an Embedded Coder license and you have selected an ERT target for your model, you can use the MATLAB command `RTW.configSubsystemBuild` to customize the generated C++ class interface to subsystem code.

C++ Class Interface Limitations

- Among the data exchange interfaces available on the **Interface** pane of the Configuration Parameters dialog box, only the C API interface is supported for C++ `class` code generation. If you select **External mode** or **ASAP2 interface**, code generation fails with a validation error.
- If a model root inport value connects to a Simscape conversion block, you must insert a Simulink Signal Conversion block between the root inport and the Simscape conversion block. On the Simulink Signal Conversion block parameter dialog box, select **Exclude this block from 'Block reduction' optimization**.
- When building a referenced model that is configured to generate a C++ class interface, you cannot use a C++ class interface in cases when a referenced model cannot have a combined output/update function. Cases include a model that
 - Has a continuous sample time
 - Saves states

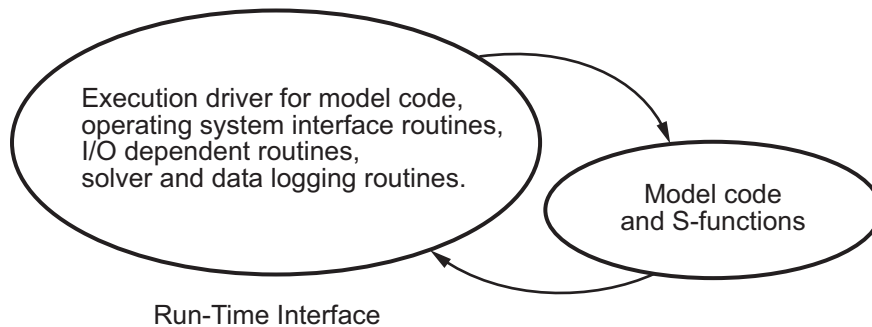
More About

- “Generate Reentrant Code from Top-Level Models” on page 15-4
- “Entry-Point Functions and Scheduling” on page 15-2
- “Execution of Code Generated from a Model” on page 15-9
- “Rapid Prototyping Model Functions” on page 15-21

Execution of Code Generated from a Model

The code generator produces algorithmic code as defined by your model. You can include external (for example, custom or legacy) code in a model by using techniques explained in “Choose External Code Integration Approach” on page 24-3.

The code generator also provides an interface that executes the generated model code. The interface and model code are compiled together to create an executable program. The next figure shows a high-level object-oriented view of the executable.



The Object-Oriented View of a Real-Time Program

In general, the conceptual design of the model execution driver does not change between the rapid prototyping and embedded style of generated code. The following sections describe model execution for single-tasking and multitasking environments both for simulation (non-real-time) and for real time. For most model code, the multitasking environment will provide the most efficient model execution (that is, fastest sample rate).

The following concepts are useful in describing how model code executes.

- **Initialization:** `model_initialize` initializes the interface code and the model code.
- **ModelOutputs:** Calls blocks in your model that have a sample hit at the current time and has them produce their output. `model_output` can be done in major or minor time steps. In major time steps, the output is a given simulation time step. In minor time steps, the interface integrates the derivatives to update the continuous states.
- **ModelUpdate:** `model_update` calls blocks in your model that have a sample hit at the current point in time and has them update their discrete states or similar type objects.

- **ModelDerivatives:** Calls blocks in your model that have continuous states and has them update their derivatives. *model_derivatives* is only called in minor time steps.
- **ModelTerminate:** *model_terminate* terminates the program if it is designed to run for a finite time. It destroys the real-time model data structure, deallocates memory, and can write data to a file.

Program Execution

A real-time program cannot require 100% of the CPU's time. This provides an opportunity to run background tasks during the free time.

Background tasks include operations such as writing data to a buffer or file, allowing access to program data by third-party data monitoring tools, or using Simulink external mode to update program parameters.

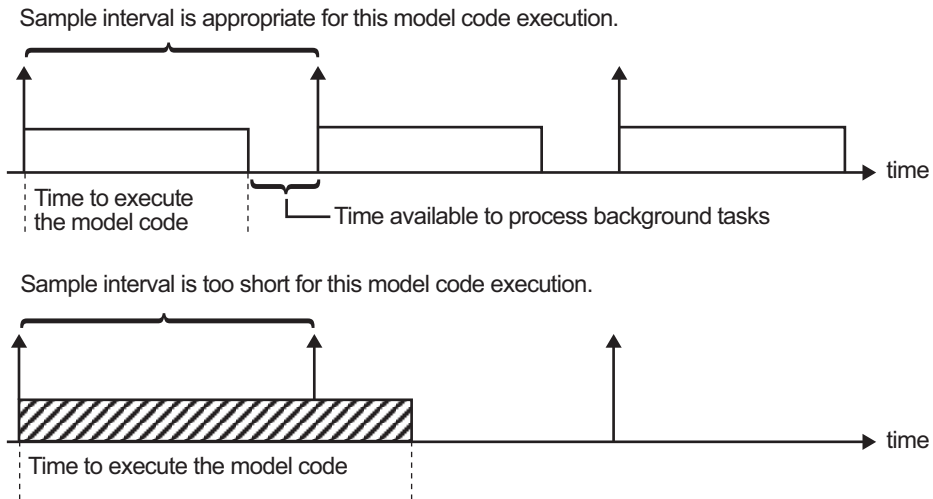
It is important, however, that the program be able to preempt the background task so the model code can execute in real time.

The way the program manages tasks depends on capabilities of the environment in which it operates.

Program Timing

Real-time programs require careful timing of the task invocations (either by using an interrupt or a real-time operating system tasking primitive) so that the model code executes to completion before another task invocation occurs. This includes time to read and write data to and from external hardware.

The next figure illustrates interrupt timing.



Task Timing

The sample interval must be long enough to allow model code execution between task invocations.

In the figure above, the time between two adjacent vertical arrows is the sample interval. The empty boxes in the upper diagram show an example of a program that can complete one step within the interval and still allow time for the background task. The gray box in the lower diagram indicates what happens if the sample interval is too short. Another task invocation occurs before the task is complete. Such timing results in an execution error.

Note also that, if the real-time program is designed to run forever (that is, the final time is 0 or infinite so that the `while` loop never exits), then the shutdown code does not execute.

For more information on how the timing engine works, see “Absolute and Elapsed Time Computation” on page 3-2.

External Mode Communication

External mode allows communication between the Simulink block diagram and the standalone program that is built from the generated code. In this mode, the real-time

program functions as an interprocess communication server, responding to requests from the Simulink engine.

Data Logging in Single-Tasking and Multitasking Model Execution

“Debug” on page 16-22 explains how you can save system states, outputs, and time to a MAT-file at the completion of the model execution. The `LogTXY` function, which performs data logging, operates differently in single-tasking and multitasking environments.

If you examine how `LogTXY` is called in the single-tasking and multitasking environments, you will notice that for single-tasking `LogTXY` is called after `ModelOutputs`. During this `ModelOutputs` call, blocks that have a hit at time t execute, whereas in multitasking, `LogTXY` is called after `ModelOutputs(tid=0)`, which executes only the blocks that have a hit at time t and that have a task identifier of 0. This results in differences in the logged values between single-tasking and multitasking logging. Specifically, consider a model with two sample times, the faster sample time having a period of 1.0 second and the slower sample time having a period of 10.0 seconds. At time $t = k*10$, $k=0,1,2,\dots$ both the fast (`tid=0`) and slow (`tid=1`) blocks execute. When executing in multitasking mode, when `LogTXY` is called, the slow blocks execute, but the previous value is logged, whereas in single-tasking the current value is logged.

Another difference occurs when logging data in an enabled subsystem. Consider an enabled subsystem that has a slow signal driving the enable port and fast blocks within the enabled subsystem. In this case, the evaluation of the enable signal occurs in a slow task, and the fast blocks see a delay of one sample period; thus the logged values will show these differences.

To summarize differences in logged data between single-tasking and multitasking, differences will be seen when

- Any root output block has a sample time that is slower than the fastest sample time
- Any block with states has a sample time that is slower than the fastest sample time
- Any block in an enabled subsystem where the signal driving the enable port is slower than the rate of the blocks in the enabled subsystem

For the first two cases, even though the logged values are different between single-tasking and multitasking, the model results are not different. The only real difference is where (at what point in time) the logging is done. The third (enabled subsystem) case results in a delay that can be seen in a real-time environment.

Non-Real-Time Single-Tasking Systems

The pseudocode below shows the execution of a model for a non-real-time single-tasking system.

```
main()
{
  Initialization
  While (time < final time)
    ModelOutputs      -- Major time step.
    LogTXY            -- Log time, states and root outputs.
    ModelUpdate       -- Major time step.
    Integrate         -- Integration in minor time step for
                    -- models with continuous states.
    ModelDerivatives
    Do 0 or more
      ModelOutputs
      ModelDerivatives
    EndDo -- Number of iterations depends upon the solver
    Integrate derivatives to update continuous states.
  EndIntegrate
EndWhile
Termination
}
```

The initialization phase begins first. This consists of initializing model states and setting up the execution engine. The model then executes, one step at a time. First `ModelOutputs` executes at time t , then the workspace I/O data is logged, and then `ModelUpdate` updates the discrete states. Next, if your model has continuous states, `ModelDerivatives` integrates the continuous states' derivatives to generate the states for time $t_{new} = t + h$, where h is the step size. Time then moves forward to t_{new} and the process repeats.

During the `ModelOutputs` and `ModelUpdate` phases of model execution, only blocks that reach the current point in time execute.

Non-Real-Time Multitasking Systems

The pseudocode below shows the execution of a model for a non-real-time multitasking system.

```
main()
```

```
{
  Initialization
  While (time < final time)
    ModelOutputs(tid=0) -- Major time step.
    LogTTY             -- Log time, states, and root
                      -- outports.
    ModelUpdate(tid=0) -- Major time step.
    Integrate          -- Integration in minor time step for
                      -- models with continuous states.
    ModelDerivatives
    Do 0 or more
      ModelOutputs(tid=0)
      ModelDerivatives
    EndDo (Number of iterations depends upon the solver.)
    Integrate derivatives to update continuous states.
  EndIntegrate
  For i=1:NumTids
    ModelOutputs(tid=i) -- Major time step.
    ModelUpdate(tid=i) -- Major time step.
  EndFor
EndWhile
Termination
}
```

Multitasking operation is more complex than single-tasking execution because the output and update functions are subdivided by the *task identifier* (`tid`) that is passed into these functions. This allows for multiple invocations of these functions with different task identifiers using overlapped interrupts, or for multiple tasks when using a real-time operating system. In simulation, multiple tasks are emulated by executing the code in the order that would occur if no preemption existed in a real-time system.

Multitasking execution assumes that all task rates are multiples of the base rate. The Simulink product enforces this when you create a fixed-step multitasking model. The multitasking execution loop is very similar to that of single-tasking, except for the use of the task identifier (`tid`) argument to `ModelOutputs` and `ModelUpdate`.

Note: You cannot use `tid` values from code generated by a target file and not by Simulink Coder. Simulink Coder tracks the use of `tid` when generating code for a specific subsystem or function type. When you generate code in a target file, this argument cannot be tracked because the scope does not have subsystem or function type. Therefore, `tid` becomes an undefined variable and your target file fails to compile.

Real-Time Single-Tasking Systems

The pseudocode below shows the execution of a model in a real-time single-tasking system where the model is run at interrupt level.

```

rtOneStep()
{
  Check for interrupt overflow
  Enable "rtOneStep" interrupt
  ModelOutputs      -- Major time step.
  LogTXY           -- Log time, states and root outputs.
  ModelUpdate       -- Major time step.
  Integrate         -- Integration in minor time step for models
                   -- with continuous states.
    ModelDerivatives
    Do 0 or more
      ModelOutputs
      ModelDerivatives
    EndDo (Number of iterations depends upon the solver.)
    Integrate derivatives to update continuous states.
  EndIntegrate
}

main()
{
  Initialization (including installation of rtOneStep as an
  interrupt service routine, ISR, for a real-time clock).
  While(time < final time)
    Background task.
  EndWhile
  Mask interrupts (Disable rtOneStep from executing.)
  Complete any background tasks.
  Shutdown
}

```

Real-time single-tasking execution is very similar to non-real-time single-tasking execution, except that instead of free-running the code, the `rt_OneStep` function is driven by a periodic timer interrupt.

At the interval specified by the program's base sample rate, the interrupt service routine (ISR) preempts the background task to execute the model code. The base sample rate is the fastest in the model. If the model has continuous blocks, then the integration step size determines the base sample rate.

For example, if the model code is a controller operating at 100 Hz, then every 0.01 seconds the background task is interrupted. During this interrupt, the controller reads its inputs from the analog-to-digital converter (ADC), calculates its outputs, writes these outputs to the digital-to-analog converter (DAC), and updates its states. Program control then returns to the background task. All of these steps must occur before the next interrupt.

Real-Time Multitasking Systems

The following pseudocode shows how a model executes in a real-time multitasking system where the model is run at interrupt level.

```
rtOneStep()
{
    Check for interrupt overflow
    Enable "rtOneStep" interrupt
    ModelOutputs(tid=0)    -- Major time step.
    LogTXY                -- Log time, states and root outputs.
    ModelUpdate(tid=0)    -- Major time step.
    Integrate              -- Integration in minor time step for
                          -- models with continuous states.

    ModelDerivatives
    Do 0 or more
        ModelOutputs(tid=0)
        ModelDerivatives
    EndDo (Number of iterations depends upon the solver.)
    Integrate derivatives and update continuous states.
EndIntegrate
For i=1:NumTasks
    If (hit in task i)
        ModelOutputs(tid=i)
        ModelUpdate(tid=i)
    EndIf
EndFor
}

main()
{
    Initialization (including installation of rtOneStep as an
        interrupt service routine, ISR, for a real-time clock).
    While(time < final time)
        Background task.
    EndWhile
}
```

```

    Mask interrupts (Disable rtOneStep from executing.)
    Complete any background tasks.
    Shutdown
}

```

Running models at interrupt level in a real-time multitasking environment is very similar to the previous single-tasking environment, except that overlapped interrupts are employed for concurrent execution of the tasks.

The execution of a model in a single-tasking or multitasking environment when using real-time operating system tasking primitives is very similar to the interrupt-level examples discussed above. The pseudocode below is for a single-tasking model using real-time tasking primitives.

```

tSingleRate()
{
    MainLoop:
        If clockSem already "given", then error out due to overflow.
        Wait on clockSem
        ModelOutputs          -- Major time step.
        LogTXY                -- Log time, states and root
                             -- outputs
        ModelUpdate           -- Major time step
        Integrate             -- Integration in minor time step
                             -- for models with continuous
                             -- states.

        ModelDeriviatives
        Do 0 or more
            ModelOutputs
            ModelDerivatives
        EndDo (Number of iterations depends upon the solver.)
        Integrate derivatives to update continuous states.
    EndIntegrate
EndMainLoop
}

main()
{
    Initialization
    Start/spawn task "tSingleRate".
    Start clock that does a "semGive" on a clockSem semaphore.
    Wait on "model-running" semaphore.
    Shutdown
}

```

In this single-tasking environment, the model executes as real-time operating system tasking primitives. In this environment, create a single task (`tSingleRate`) to run the model code. This task is invoked when a clock tick occurs. The clock tick gives a `clockSem` (clock semaphore) to the model task (`tSingleRate`). The model task waits for the semaphore before executing. The clock ticks occur at the fundamental step size (base rate) for your model.

Multitasking Systems Using Real-Time Tasking Primitives

The pseudocode below is for a multitasking model using real-time tasking primitives.

```
tSubRate(subTaskSem,i)
{
  Loop:
    Wait on semaphore subTaskSem.
    ModelOutputs(tid=i)
    ModelUpdate(tid=i)
  EndLoop
}
tBaseRate()
{
  MainLoop:
    If clockSem already "given", then error out due to overflow.
    Wait on clockSem
    For i=1:NumTasks
      If (hit in task i)
        If task i is currently executing, then error out due to
        overflow.
        Do a "semGive" on subTaskSem for task i.
      EndIf
    EndFor
    ModelOutputs(tid=0)    -- major time step.
    LogTXY                 -- Log time, states and root outputs.
    ModelUpdate(tid=0)    -- major time step.
    Loop:                 -- Integration in minor time step for
                        -- models with continuous states.
      ModelDerivatives
      Do 0 or more
        ModelOutputs(tid=0)
        ModelDerivatives
      EndDo (number of iterations depends upon the solver).
      Integrate derivatives to update continuous states.
    EndLoop
}
```

```

    EndMainLoop
}
main()
{
    Initialization
    Start/spawn task "tSubRate".
    Start/spawn task "tBaseRate".

    Start clock that does a "semGive" on a clockSem semaphore.
    Wait on "model-running" semaphore.
    Shutdown
}

```

In this multitasking environment, the model is executed using real-time operating system tasking primitives. Such environments require several model tasks (`tBaseRate` and several `tSubRate` tasks) to run the model code. The base rate task (`tBaseRate`) has a higher priority than the subrate tasks. The subrate task for `tid=1` has a higher priority than the subrate task for `tid=2`, and so on. The base rate task is invoked when a clock tick occurs. The clock tick gives a `clockSem` to `tBaseRate`. The first thing `tBaseRate` does is give semaphores to the subtasks that have a hit at the current point in time. Because the base rate task has a higher priority, it continues to execute. Next it executes the fastest task (`tid=0`), consisting of blocks in your model that have the fastest sample time. After this execution, it resumes waiting for the clock semaphore. The clock ticks are configured to occur at the fundamental step size for your model.

Rapid Prototyping and Embedded Model Execution Differences

The rapid prototyping program framework provides a common application programming interface (API) that does not change between model definitions.

The Embedded Coder product provides a different framework called the embedded program framework. The embedded program framework provides an optimized API that is tailored to your model. When you use the embedded style of generated code, you are modeling how you would like your code to execute in your embedded system. Therefore, the definitions defined in your model should be specific to your embedded targets. Items such as the model name, parameter, and signal storage class are included as part of the API for the embedded style of code.

One major difference between the rapid prototyping and embedded style of generated code is that the latter contains fewer entry-point functions. The embedded style of code can be configured to have only one function, `model_step`.

Thus, model execution code eliminates `Loop...EndLoop` statements and groups `ModelOutputs`, `LogTXY`, and `ModelUpdate` into a single statement, `model_step`.

For more information about how generated embedded code executes, see “Entry-Point Functions and Scheduling” on page 15-2.

More About

- “Time-Based Scheduling and Code Generation”
- “Sample Times in Subsystems”
- “Sample Times in Systems”
- “Time-Based Scheduling Example Models”

Rapid Prototyping Model Functions

Rapid prototyping code defines the following functions that interface with the main program (`main.c` or `main.cpp`):

- `Model()`: The model registration function. This function initializes the work areas (for example, allocating and setting pointers to various data structures) used by the model. The model registration function calls the `MdlInitializeSizes` and `MdlInitializeSampleTimes` functions. These two functions are very similar to the S-function `mdlInitializeSizes` and `mdlInitializeSampleTimes` methods.
- `MdlStart(void)`: After the model registration functions `MdlInitializeSizes` and `MdlInitializeSampleTimes` execute, the main program starts execution by calling `MdlStart`. This routine is called once at startup.

The function `MdlStart` has four basic sections:

- Code to initialize the states for each block in the root model that has states. A subroutine call is made to the “initialize states” routines of conditionally executed subsystems.
- Code generated by the one-time initialization (start) function for each block in the model.
- Code to enable the blocks in the root model that have enable methods, and the blocks inside triggered or function-call subsystems residing in the root model. Simulink blocks can have enable and disable methods. An enable method is called just before a block starts executing, and the disable method is called just after the block stops executing.
- Code for each block in the model whose output value is constant. The block code appears in the `MdlStart` function only if the block parameters are not tunable in the generated code and if the code generator cannot eliminate the block code through constant folding.
- `MdlOutputs(int_T tid)`: `MdlOutputs` updates the output of blocks. The `tid` (task identifier) parameter identifies the task that in turn maps when to execute blocks based upon their sample time. This routine is invoked by the main program during major and minor time steps. The major time steps are when the main program is taking an actual time step (that is, it is time to execute a specific task). If your model contains continuous states, the minor time steps will be taken. The minor time steps are when the solver is generating integration stages, which are points between major outputs. These integration stages are used to compute the derivatives used in advancing the continuous states.

- `MdlUpdate(int_T tid)`: `MdlUpdate` updates the states and work vector state information (that is, states that are neither continuous nor discrete) saved in work vectors. The `tid` (task identifier) parameter identifies the task that in turn indicates which sample times are active, allowing you to conditionally update only states of active blocks. This routine is invoked by the interface after the major `MdlOutputs` has been executed. The solver is also called, and `model_Derivatives` is called in minor steps by the solver during its integration stages. All blocks that have continuous states have an identical number of derivatives. These blocks are required to compute the derivatives so that the solvers can integrate the states.
- `MdlTerminate(void)`: `MdlTerminate` contains any block shutdown code. `MdlTerminate` is called by the interface, as part of the termination of the real-time program.

The contents of the above functions are directly related to the blocks in your model. A Simulink block can be generalized to the following set of equations.

$$y = f_0(t, x_c, x_d, u)$$

Output y is a function of continuous state x_c , discrete state x_d , and input u . Each block writes its specific equation in a section of `MdlOutputs`.

$$x_{d+1} = f_u(t, x_d, u)$$

The discrete states x_d are a function of the current state and input. Each block that has a discrete state updates its state in `MdlUpdate`.

$$\dot{x} = f_d(t, x_c, u)$$

The derivatives x are a function of the current input. Each block that has continuous states provides its derivatives to the solver (for example, `ode5`) in `model_Derivatives`. The derivatives are used by the solver to integrate the continuous state to produce the next value.

The output, y , is generally written to the block I/O structure. Root-level Output blocks write to the external outputs structure. The continuous and discrete states are stored in the states structure. The input, u , can originate from another block's output, which is located in the block I/O structure, an external input (located in the external inputs

structure), or a state. These structures are defined in the *model.h* file that the Simulink Coder software generates.

The next example shows the general contents of the rapid prototyping style of C code written to the *model.c* file.

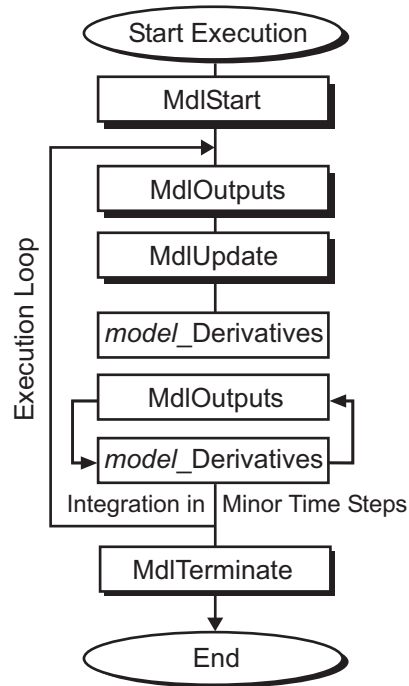
```
/*
 * Version, Model options, TLC options,
 * and code generation information are placed here.
 */
<includes>
void MdlStart(void)
{
    /*
     * State initialization code.
     * Model start-up code - one time initialization code.
     * Execute any block enable methods.
     * Initialize output of any blocks with constant sample times.
     */
}

void MdlOutputs(int_T tid)
{
    /* Compute: y = f0(t,xc,xd,u) for each block as needed. */
}

void MdlUpdate(int_T tid)
{
    /* Compute: xd+1 = fu(t,xd,u) for each block as needed. */

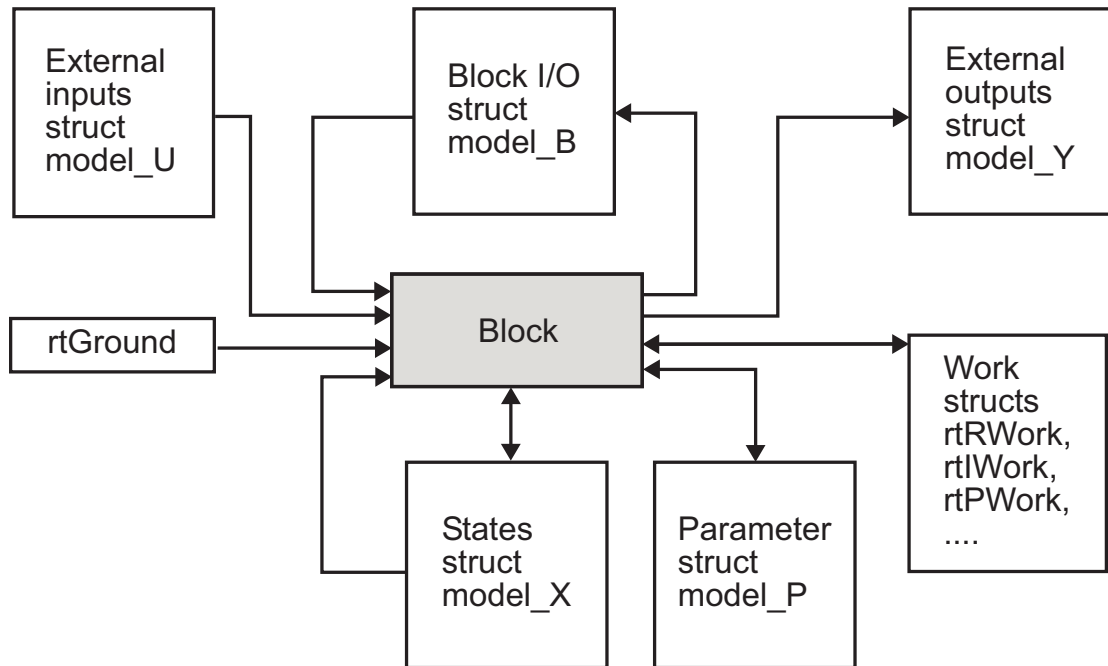
    /* Compute: dxc = fd(t,xc,u) for each block in model_derivatives
       as needed. */
}
void MdlTerminate(void)
{
    /* Perform shutdown code for any blocks that
       have a termination action */
}
```

The next figure shows a flow chart describing the execution of the rapid prototyping generated code.



Rapid Prototyping Execution Flow Chart

Each block places code in specific Mdl routines according to the algorithm that it is implementing. Blocks have input, output, parameters, and states, as well as other general items. For example, in general, block inputs and outputs are written to a block I/O structure (*model_B*). Block inputs can also come from the external input structure (*model_U*) or the state structure when connected to a state port of an integrator (*model_X*), or ground (`rtGround`) if unconnected or grounded. Block outputs can also go to the external output structure (*model_Y*). The next figure shows the general mapping between these items.



Data View of the Generated Code

The following list defines the structures shown in the preceding figure:

- Block I/O structure (*model_B*): This structure consists of persistent block output signals. The number of block output signals is the sum of the widths of the data output ports of all nonvirtual blocks in your model. If you activate block I/O optimizations, the Simulink and Simulink Coder products reduce the size of the *model_B* structure by
 - Reusing the entries in the *model_B* structure
 - Making other entries local variables

See “Signal Representation in Generated Code” on page 14-109 for more information on these optimizations.

Structure field names are determined either by the block’s output signal name (when present) or by the block name and port number when the output signal is left unlabeled.

- Block states structures: The continuous states structure (*model_X*) contains the continuous state information for blocks in your model that have continuous states. Discrete states are stored in a data structure called the *DWork vector* (*model_DWork*).
- Block parameters structure (*model_P*): The parameters structure contains block parameters that can be changed during execution (for example, the parameter of a Gain block).
- External inputs structure (*model_U*): The external inputs structure consists of all root-level Inport block signals. Field names are determined by either the block's output signal name, when present, or by the Inport block's name when the output signal is left unlabeled.
- External outputs structure (*model_Y*): The external outputs structure consists of all root-level Outport blocks. Field names are determined by the root-level Outport block names in your model.
- Real work, integer work, and pointer work structures (*model_RWork*, *model_IWork*, *model_PWork*): Blocks might have a need for real, integer, or pointer work areas. For example, the Memory block uses a real work element for each signal. These areas are used to save internal states or similar information.

More About

- “Time-Based Scheduling and Code Generation”

Code Generation

Configuration for Simulink Coder

- “Code Generation Configuration” on page 16-2
- “Configure Code Generation Parameters for Model Programmatically” on page 16-4
- “Check Model and Configuration for Code Generation” on page 16-6
- “Application Objectives Using Code Generation Advisor” on page 16-8
- “Simulink Coder Model Advisor Checks for Standards and Code Efficiency” on page 16-12
- “Configure Code Comments” on page 16-13
- “Construction of Generated Identifiers” on page 16-14
- “Identifier Name Collisions and Mangling” on page 16-15
- “Specify Identifier Length to Avoid Naming Collisions” on page 16-16
- “Specify Reserved Names for Generated Identifiers” on page 16-17
- “Reserved Keywords” on page 16-18
- “Debug” on page 16-22

Code Generation Configuration

When you are ready to generate code for a model, you can modify the model configuration parameters specific to code generation. The code generation parameters determine how the code generator produces code and builds an executable program from your model. You modify the model configuration parameters for code generation in the **Code Generation** and **Optimization** panes of the Configuration Parameters dialog box. The content of the **Code Generation** pane and its subpanes can change depending on the target that you specify. Some configuration options are available only with the Embedded Coder product. The **Optimization** pane includes code generation parameters that help to improve the performance of the generated code. To open the **Code Generation** pane, see “Open the Model Configuration for Code Generation” on page 16-2.

Your application objectives might include a combination of the following code generation objectives: debugging, traceability, execution efficiency, and safety precaution. There are tradeoffs associated with these configuration choices, such as execution speed and memory usage. You can use the Model Advisor and the Code Generation Advisor to help configure a model to achieve your application objectives. For more information, see “Application Objectives Using Code Generation Advisor” on page 16-8.

To automate the configuration of models, you can use the `set_param` function from the MATLAB command line to adjust the model configuration parameters. For more information, see “Configure Code Generation Parameters for Model Programmatically” on page 16-4.

Open the Model Configuration for Code Generation

To modify the model configuration parameters for code generation, open the **Code Generation** pane. There are four ways to open the **Code Generation** pane from the Simulink editor:

- To open the Configuration Parameters dialog box, click the model configuration parameters icon:



and select **Code Generation**.

- From the **Simulation** menu, select **Model Configuration Parameters**. When the Configuration Parameters dialog box opens, click **Code Generation** in the **Select** (left) pane.
- From the Code menu, select **C/C++ Code > Code Generation Options**.
- From the **View** menu in the model window, select **Model Explorer**, or from the MATLAB command line, type `daexplr` and press **Enter**. In the Model Explorer, expand the node for the current model in the left pane and click **Configuration (active)**. The configuration elements are listed in the middle pane. Clicking one of these elements displays the corresponding parameters in the right pane. Alternatively, right-clicking the **Code Generation** element in the middle pane and choosing **Properties** from the context menu opens the **Code Generation** pane in a separate window.

Note In a Configuration Parameters dialog box, when you change a check box, menu selection, or edit field, the white background of the element turns to light yellow to indicate that you made an unsaved change. When you click **OK**, **Cancel**, or **Apply**, the background resets to white.

For more information on model configurations, see “Configuration Reuse”.

Configure Code Generation Parameters for Model Programmatically

This example shows how to modify code generation parameters for the active configuration set from the MATLAB command line. Use this approach for creating a script that automates setting parameters for an established model configuration. You can use the Configuration Parameters **All Parameters** tab to get a parameter's command line name to use in scripts. In this example, you modify the configuration parameters to support the Code Generation Advisor application objective, **Execution efficiency**.

Step 1. Open a model.

```
slexAircraftExample
```

Step 2. Get the active configuration set.

```
cs = getActiveConfigSet(model);
```

Step 3. Select the Generic Real-Time (GRT) target.

```
switchTarget(cs, 'grt.tlc', []);
```

Step 4. Modify parameters to optimize execution speed.

If your application objective is **Execution efficiency**, use `set_param` to modify the following parameters:

```
set_param(cs, 'MatFileLogging', 'off');  
set_param(cs, 'SupportNonFinite', 'off');  
set_param(cs, 'RTWCompilerOptimization', 'on');  
set_param(cs, 'OptimizeBlockIOStorage', 'on');  
set_param(cs, 'EnhancedBackFolding', 'on');  
set_param(cs, 'ConditionallyExecuteInputs', 'on');  
set_param(cs, 'DefaultParameterBehavior', 'Inlined');  
set_param(cs, 'BooleanDataType', 'on');  
set_param(cs, 'BlockReduction', 'on');  
set_param(cs, 'ExpressionFolding', 'on');  
set_param(cs, 'LocalBlockOutputs', 'on');  
set_param(cs, 'EfficientFloat2IntCast', 'on');  
set_param(cs, 'BufferReuse', 'on');
```

Step 5. Save the model configuration to a file.

Save the model configuration to a file, 'Exec_efficiency_cs.m', and view the parameter settings.

```
saveAs(cs, 'Exec_Efficiency_cs');  
dbtype Exec_Efficiency_cs 1:50
```

More About

- “Configuration Parameter Basics”

Check Model and Configuration for Code Generation

You can use the Model Advisor checks to assess model readiness to generate code. To check and configure your model for code generation application objectives such as traceability or debugging, use the Code Generation Advisor.

For information about	See
Model Advisor	“Run Model Checks”
Code Generation Advisor	“Application Objectives Using Code Generation Advisor” on page 16-8
Checks available with Simulink Coder	“Simulink Coder Checks”
Checks available with Embedded Coder	“Embedded Coder Checks”

This example shows how to use the Model Advisor to check model `rtwdemo_throttlecntrl` for code efficiency.

- 1 Open `rtwdemo_throttlecntrl`. Save a copy as `throttlecntrl` in a writable location on your MATLAB path.
- 2 To start the Model Advisor, select **Analysis > Model Advisor > Model Advisor**. A dialog box opens showing the model system hierarchy.
- 3 Click `throttlecntrl` and then click **OK**. The Model Advisor window opens.
- 4 Expand **By Task > Code Generation Efficiency**. You can use the checks in the folder to check your model for code generation efficiency. By default, checks that do not trigger an Update Diagram are selected.

Checks for code generation efficiency depend on the availability of Simulink Coder and Embedded Coder licenses.

- 5 In the left pane, select the remaining checks, and then select **Code Generation Efficiency**.
- 6 In the right pane, select **Show report after run** and click **Run Selected Checks**. The report shows a **Run Summary** that flags check warnings.
- 7 Review the report. The warnings highlight issues that impact code efficiency. For more information about the report, see “View Model Advisor Reports” in the Simulink documentation.

Check Model During Code Generation

This example shows how to use the Code Generation Advisor to review a model as part of the code generation process.

- 1 Specify your code generation objectives.
- 2 On the **Configuration Parameters > Code Generation > General** pane, select one of the following from **Check model before generating code**:
 - On (proceed with warnings)
 - On (stop for warnings)
- 3 If you only want to generate code, select **Generate code only**; otherwise clear the check box to build an executable.
- 4 Apply your changes, and then click **Generate Code/Build**. The Code Generation Advisor starts and reviews the top model and subsystems.

If the Code Generation Advisor issues failures or warnings, and you specified:

- **On (proceed with warnings)** — The Code Generation Advisor window opens while the build process proceeds. After the build process is complete, you can review the results.
 - **On (stop for warnings)** — The build process halts and displays the Diagnostic Viewer. To continue, you must review and resolve the Code Generation Advisor results or change the **Check model before generating code** selection.
- 5 In the Code Generation Advisor window, review the results by selecting a check from the left pane. The right pane populates the results for that check.
 - 6 After reviewing the check results, you can choose to fix warnings and failures as described in “Fix a Model Check Warning or Failure”.

Note: When you specify an efficiency or Safety precaution objective, the Code Generation Advisor includes additional checks. When you make changes to one of these checks, the other check results could become invalid and you must run the check again.

For more information, see “Set Objectives — Code Generation Advisor Dialog Box”

Application Objectives Using Code Generation Advisor

In this section...

“High-Level Code Generation Objectives” on page 16-9

“Configure Model for Code Generation Objectives Using Code Generation Advisor” on page 16-10

“Configure Model for Code Generation Objectives Using Configuration Parameters Dialog Box” on page 16-11

Consider how your application objectives, such as efficiency, traceability, and safety, map to code generation options in a model configuration set. Parameters that you set in the **Solver**, **Data Import/Export**, **Diagnostics**, and **Code Generation** panes of the Configuration Parameters dialog box specify the behavior of a model in simulation and the code generated for the model.

You can use the Code Generation Advisor to review a model before generating code, or as part of the code generation process. When you choose to review a model before generating code, you specify which model, subsystem, or referenced model the Code Generation Advisor reviews. When you choose to review a model as part of the code generation process, the Code Generation Advisor reviews the entire system. The Code Generation Advisor uses the information presented in “Recommended Settings Summary for Model Configuration Parameters” to determine the values. When there is a conflict due to multiple objectives, the higher-priority objective takes precedence.

Setting code generation objectives and running the Code Generation Advisor provides information on how to meet code generation objectives for your model. The Code Generation Advisor does not alter the generated code. You can use the Code Generation Advisor to make the suggested changes to your model. The generated code is changed only after you modify your model and regenerate code. If you use the Code Generation Advisor to set code generation objectives and check your model, the generated code includes comments identifying which objectives you specified, the checks the Code Generation Advisor ran on the model, and the results of running the checks.

If a model uses a configuration reference, you can run the Code Generation Advisor to review your configuration parameter settings, but the Code Generation Advisor cannot modify the configuration parameter settings.

High-Level Code Generation Objectives

Depending on the type of application that your model represents, you are likely to have specific high-level code generation objectives. For example, safety and traceability might be more critical than efficient use of memory. If you have specific objectives, you can quickly configure your model to meet those objectives by selecting and prioritizing from these code generation objectives:


- Execution efficiency (all targets) — Configure code generation settings to achieve fast execution time.
- ROM efficiency (ERT-based targets) — Configure code generation settings to reduce ROM usage.
- RAM efficiency (ERT-based targets) — Configure code generation settings to reduce RAM usage.
- Traceability (ERT-based targets) — Configure code generation settings to provide mapping between model elements and code.
- Safety precaution (ERT-based targets) — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.
- Debugging (all targets) — Configure code generation settings to debug the code generation build process.
- MISRA C:2012 guidelines (ERT-based targets) — Configure code generation settings to increase compliance with MISRA C:2012 guidelines.
- Polyspace (ERT-based targets) — Configure code generation settings to prepare the code for Polyspace[®] analysis.

Note: If you select the MISRA C:2012 guidelines code generation objective, the Code Generation Advisor checks:

- The model configuration settings for compliance with the MISRA C:2012 configuration setting recommendations.
 - For blocks that are not supported or recommended for MISRA C:2012 compliant code generation.
-

Configure Model for Code Generation Objectives Using Code Generation Advisor

This example shows how to use the Code Generation Advisor to check and configure your model to meet code generation objectives:

- 1 On the menu bar, select **Code > C/C++ Code > Code Generation Advisor**.
Alternatively:
 - On the toolbar  drop-down list, select **Code Generation Advisor**.
 - Right-click a subsystem, and then select **C/C++ Code > Code Generation Advisor**. Go to step 3.
- 2 In the System Selector window, select the model or subsystem that you want to review, and then click **OK**.
- 3 In the Code Generation Advisor, on the **Code Generation Objectives** pane, select the code generation objectives from the drop-down list (GRT-based targets). As you select objectives, on the left pane, the Code Generation Advisor updates the list of checks it will run on your model. If your model is configured with an ERT-based target, more objectives are available.
- 4 Click **Run Selected Checks** to run the checks listed in the left pane of the Code Generation Advisor.
- 5 In the Code Generation Advisor window, review the results for **Check model configuration settings against code generation objectives** by selecting it from the left pane. The results for that check are displayed in the right pane.

Check model configuration settings against code generation objectives triggers a warning for either of these reasons:

- Parameters are set to values other than the value recommended for the specified code generation objectives.
- Selected code generation objectives differ from the objectives set in the model.

Click **Modify Parameters** to set:

- Parameter to the value recommended for the specified code generation objectives.
- Code generation objectives in the model to the objectives specified in the Code Generation Advisor.

- 6 In the Code Generation Advisor window, review the results for the remaining checks by selecting them from the left pane. The results for that check are displayed in the right pane.
- 7 After reviewing the check results, you can choose to fix warnings and failures, as described in “Fix a Model Check Warning or Failure”.

Note: When you specify an efficiency or safety precaution objective, the Code Generation Advisor includes additional checks.

When you make changes to one check, the other check results are not valid. You must run the checks again.

Configure Model for Code Generation Objectives Using Configuration Parameters Dialog Box

This example shows how to check and configure the code generation objectives in the Configuration Parameters dialog box:

- 1 Open the Configuration Parameters dialog box and select **Code Generation**.
- 2 Select or confirm selection of a System target file.
- 3 Specify the objectives using the **Select objectives** drop down list (GRT-based targets) or clicking **Set Objectives** button (ERT-based targets). Clicking **Set Objectives** opens the “Set Objectives — Code Generation Advisor Dialog Box” dialog box.
- 4 Click **Check Model** to run the model checks. The Code Generation Advisor dialog box opens. The Code Generation Advisor uses the code generation objectives to determine which model checks to run.
- 5 On the left pane, the Code Generation Advisor lists the checks run on the model and the results. Click each warning to see the suggestions for changes that you can make to your model to pass the check.
- 6 Determine which changes to make to your model. On the right pane of the Code Generation Advisor, follow the instructions listed for each check to modify the model.

Simulink Coder Model Advisor Checks for Standards and Code Efficiency

You can use the Model Advisor checks available with Simulink Coder to check that your model meets standards and is ready to generate code.

- To start the Model Advisor, in the model window, select **Analysis > Model Advisor > Model Advisor**.
- In the Model Advisor window, expand the **By Task** folder. The folder contains Model Advisor checks that you can run to help accomplish the task.

For more information about the Model Advisor, see “Run Model Checks”.

The table summarizes the Simulink Coder Model Advisor checks that are available in the **By Task** folders.

By Task folder	Model Advisor checks
Code Generation Efficiency	“Identify blocks using one-based indexing”
Modeling Standards for DO-178C/DO-331	“Check solver for code generation” “Check for blocks that have constraints on tunable parameters” “Check sample times and tasking mode”
Model Referencing	“Check for model reference configuration mismatch” “Check for code generation identifier formats used for model reference”

Configure Code Comments

Configure how the code generator inserts comments into generated code by modifying parameters on the **Code Generation > Comments** pane.

To...	Select...
Include comments in generated code	Include comments. Selecting this parameter allows you to select one or more auto generated comment types to be placed in the code.
Include comments that describe a block's code	Simulink block / Stateflow object comments. Selecting this parameter includes the comments before the block's code in the generating file.
Include MATLAB source code as comments	MATLAB source code as comments. Selecting this parameter inserts these comments preceding the associated generated code. The function signature is included in the function banner.
Include comments for eliminated blocks	Show eliminated blocks. Selecting this parameter includes comments for blocks that were eliminated as the result of optimizations, such as inlining parameters.
Include parameter comments regardless of the number of parameters	<p>Verbose comments for SimulinkGlobal storage class. Selecting this parameter includes comments for parameter variable names and names of source blocks in the model parameter structure declaration in <i>model_prm.h</i>.</p> <p>If you do not select this parameter, parameter comments are generated if less than 1000 parameters are declared. This reduces the size of the generated file for models with a large number of parameters.</p>

Note When you configure the code generator to produce code that includes comments, the code generator includes text for model parameters, block names, signal names, and Stateflow object names in the generated code comments. If the text includes characters that are unrepresented in the character set encoding for the model, the code generator replaces the characters with XML escape sequences. For example, the code generator replaces the Japanese full-width Katakana letter ア with the escape sequence `ア`. For more information, see “Internationalization and Code Generation” on page 18-2.

Construction of Generated Identifiers

For GRT and RSim targets, the code generator automatically constructs identifiers for variables and functions in the generated code. These identifiers represent:

- Signals and parameters that have `Auto` storage class
- Subsystem function names that are not user defined
- Stateflow names

The components of a generated identifier include

- The root model name, followed by
- The name of the generating object (signal, parameter, state, and so on), followed by
- Unique *name-mangling* text

The code generator conditionally generates the name-mangling text to resolve potential conflicts with other generated identifiers.

To configure how the code generator names identifiers and objects, see:

- “Specify Identifier Length to Avoid Naming Collisions” on page 16-16
- “Specify Reserved Names for Generated Identifiers” on page 16-17

The code generator reserves certain words for its own use as keywords of the generated code language. For more information, see “Reserved Keywords” on page 16-18.

With an Embedded Coder license, you can specify parameters to control identifier formats, mangle length, scalar inlined parameters, and Simulink data object naming rules. For more information, see “Customize Generated Identifier Naming Rules”.

Identifier Name Collisions and Mangling

In identifier generation, a circumstance that would cause generation of two or more identical identifiers is called a *name collision*. When a potential name collision exists, unique *name-mangling* text is generated and inserted into each of the potentially conflicting identifiers. Each set of name-mangling characters is unique for each generated identifier.

Identifier Name Collisions with Referenced Models

Referenced models can introduce additional naming constraints. Within a model that uses referenced models, collisions between the names of the models cannot exist. When you generate code from a model that includes referenced models, the **Maximum identifier length** parameter must be large enough to accommodate the root model name and name-mangling text. A code generation error occurs if **Maximum identifier length** is too small.

When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the identifier from the referenced model is preserved. Name mangling is performed on the identifier from the higher-level model.

For more information on referenced models, see “Parameterize Instances of a Reusable Referenced Model”.

Specify Identifier Length to Avoid Naming Collisions

The length of a generated identifier is limited by the **Maximum identifier length** parameter specified on the **Symbols** pane of the Configuration Parameters dialog box. The **Maximum identifier length** field allows you to limit the number of characters in function, type definition, and variable names. The default is 31 characters. This is also the minimum length you can specify. The maximum is 256 characters.

When there is a potential name collision between two identifiers, name-mangling text is generated. The text has the minimum number of characters required to avoid the collision. The other symbol components are then inserted. If **Maximum identifier length** is not large enough to accommodate full expansions of the other components, they are truncated. To avoid this outcome, it is good practice to:

- Avoid name collisions by not using default block names (for example, `Gain1`, `Gain2...`) when the model includes multiple blocks of the same type.
- For subsystems, make them atomic and reusable.
- Increase the **Maximum identifier length** parameter to accommodate the length of the identifier you expect to generate.

Specify Reserved Names for Generated Identifiers

You can specify a set of reserved keywords that the code generation process should not use, facilitating code integration where functions and variables from external environments are unknown in the Simulink model. To create a list of reserved names, open the Configuration Parameters dialog box. On the **Code Generation > Symbols** pane, enter the keywords in the “Reserved names” field.

If your model contains MATLAB Function or Stateflow blocks, the code generation process can use the reserved names specified for those blocks if you select **Use the same reserved names as Simulation Target** on the **Code Generation > Symbols** pane.

Reserved Keywords

In this section...

“C Reserved Keywords” on page 16-18

“C++ Reserved Keywords” on page 16-19

“Reserved Keywords for Code Generation” on page 16-19

“Code Generation Code Replacement Library Keywords” on page 16-20

Generator keywords are reserved for internal use. Do not use them in models as identifiers or function names. Also avoid using C reserved keywords in models as identifiers or function names. If your model contains reserved keywords, code generation does not complete and an error message appears. To address the error, modify your model to use identifiers or names that are not reserved.

If you use the code generator to produce C++ code, your model must not contain the “Reserved Keywords for Code Generation” on page 16-19 nor the “C++ Reserved Keywords” on page 16-19.

Note: You can register additional reserved identifiers in the Simulink environment. For more information, see “Specify Reserved Names for Generated Identifiers” on page 16-17.

C Reserved Keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

C++ Reserved Keywords

catch	friend	protected	try
class	inline	public	typeid
const_cast	mutable	reinterpret_cast	typename
delete	namespace	static_cast	using
dynamic_cast	new	template	virtual
explicit	operator	this	wchar_t
export	private	throw	

Reserved Keywords for Code Generation

abs	int_T	MAX_int64_T	RT_MALLOC
asm	int8_T	MAX_uint8_T*	rtInf
bool	int16_T	MAX_uint16_T*	rtMinusInf
boolean_T	int32_T	MAX_uint32_T*	rtNaN
byte_T	int64_T	MAX_uint64_T	SeedFileBuffer
char_T	INTEGER_CODE	MIN_int8_T*	SeedFileBufferLen
cint8_T	LINK_DATA_BUFFER_S	MIN_int16_T*	single
cint16_T	LINK_DATA_STREAM	MIN_int32_T*	TID01EQ
cint32_T	localB	MIN_int64_T	time_T
creal_T	localC	MODEL	true
creal32_T	localDWork	MT	TRUE
creal64_T	localP	NCSTATES	uint_T
cuint8_T	localX	NULL	uint8_T
cuint16_T	localXdis	NUMST	uint16_T
cuint32_T	localXdot	pointer_T	uint32_T
ERT	localZCE	PROFILING_ENABLED	uint64_T
false	localZCSV	PROFILING_NUM_SAMP	UNUSED_PARAMETER
FALSE	matrix	real_T	USE_RTMODEL

fortran	MAX_int8_T*	real32_T	VCAST_FLUSH_DATA
HAVESTDIO	MAX_int16_T*	real64_T	vector
id_t	MAX_int32_T*	RT	
*Not reserved if you specify a replacement identifier.			

Code Generation Code Replacement Library Keywords

The list of code replacement library reserved keywords for your development environment varies depending on which libraries are registered. The list of available code replacement libraries varies depending on other installed products (for example, a target product), or if you used Embedded Coder to create and register custom code replacement libraries.

To generate a list of reserved keywords for libraries currently registered in your environment, use the following MATLAB function:

```
lib_ids = RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers()
```

This function returns an array of library keywords. Specifying the input argument is optional.

Note: To list the libraries currently registered in your environment, use the MATLAB command `crviewer`.

To generate a list of reserved keywords for a specific library that you are using to generate code, call the function passing the name of the library as displayed in the **Code replacement library** menu on the **Code Generation > Interface** pane of the Configuration Parameters dialog box. For example,

```
lib_ids = RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers('GNU C99 Extensions')
```

Here is a partial example of the function output:

```
>> lib_ids = RTW.TargetRegistry.getInstance.getTf1ReservedIdentifiers('GNU C99 Extensions')
lib_ids =
    'exp10'
    'exp10f'
    'acosf'
    'acoshf'
```

```
'asinf'  
'asinhf'  
'atanf'  
'atanhf'  
...  
'rt_lu_cplx'  
'rt_lu_cplx_sgl'  
'rt_lu_real'  
'rt_lu_real_sgl'  
'rt_mod_boolean'  
'rt_rem_boolean'  
'strcpy'  
'utAssert'
```

Note: Some of the returned keywords appear with the suffix \$N, for example, 'rt_atan2\$N'. \$N expands into the suffix _snf only if nonfinite numbers are supported. For example, 'rt_atan2\$N' represents 'rt_atan2_snf' if nonfinite numbers are supported and 'rt_atan2' if nonfinite numbers are not supported. As a precaution, you should treat both forms of the keyword as reserved.

Debug

In the Configuration Parameters dialog box, use parameters on the **Diagnostics** pane and debugging parameters on the **All Parameters** tab to configure a model such that the generated code and the build process are set for debugging. You can set parameters that apply to the model compilation phase, the target language code generation phase, or both.

Parameters in the following table will be helpful if you are writing TLC code for customizing targets, integrating legacy code, or developing new blocks.

To...	Select...
Display progress information during code generation in the MATLAB Command Window	“Verbose build”. Compiler output also displays.
Prevent the build process from deleting the <i>model.rtw</i> file from the build folder at the end of the build	“Retain .rtw file”. This parameter is useful if you are modifying the target files, in which case you need to look at the <i>model.rtw</i> file.
Instruct the TLC profiler to analyze the performance of TLC code executed during code generation and generate a report	“Profile TLC”. The report is in HTML format and can be read in your Web browser.
Start the TLC debugger during code generation	“Start TLC debugger when generating code”. Alternatively, enter the argument <code>-dc</code> for the “System target file” parameter on the Code Generation pane. To start the debugger and run a debugger script, enter <code>-df filename</code> for System target file .
Generate a report containing statistics indicating how many times the code generator reads each	“Start TLC coverage when generating code”. Alternatively, enter the argument <code>-dg</code> for the System Target File parameter on the Code Generation pane.

To...	Select...
line of TLC code during code generation	
Halt a build if a user-supplied TLC file contains an <code>%assert</code> directive that evaluates to FALSE	<p>“Enable TLC assertion”. Alternatively, you can use MATLAB commands to control TLC assertion handling.</p> <p>To set the flag on or off, use the <code>set_param</code> command. The default is off.</p> <pre>set_param(model, 'TLCAssertion', 'on off')</pre> <p>To check the current setting, use <code>get_param</code>.</p> <pre>get_param(model, 'TLCAssertion')</pre>
Detect loss of tunability	<p>“Detect loss of tunability” on the Diagnostics > Data Validity pane. You can use this parameter to report loss of tunability when an expression is reduced to a numeric expression. This can occur if a tunable workspace variable is modified by Mask Initialization code, or is used in an arithmetic expression with unsupported operators or functions. Possible values are:</p> <ul style="list-style-type: none"> • <code>none</code> — Loss of tunability can occur without notification. • <code>warning</code> — Loss of tunability generates a warning (default). • <code>error</code> — Loss of tunability generates an error. <p>For a list of supported operators and functions, see “Tunable Expression Limitations” on page 14-49</p>

To...	Select...
<p>Enable model verification (assertion) blocks</p>	<p>“Model Verification block enabling” on the All Parameters. Use this parameter to enable or disable model verification blocks such as Assert, Check Static Gap, and related range check blocks. The diagnostic applies to generated code as well as simulation behavior. For example, simulation and code generation ignore this parameter when model verification blocks are inside an S-function. Possible values are:</p> <ul style="list-style-type: none"> • User local settings • Enable All • Disable All <p>For Assertion blocks not disabled, generated code for a model includes one of the following statements, depending on the blocks input signal type (Boolean, real, or integer, respectively).</p> <pre>utAssert(input_signal); utAssert(input_signal != 0.0); utAssert(input_signal != 0);</pre> <p>By default, <code>utAssert</code> does not change generated code. For assertions to abort execution, you must enable them by specifying the following <code>make_rtw</code> command for Code Generation > “Make command” parameter:</p> <pre>make_rtw OPTS="-DDOASSERTS"</pre> <p>Use the following variant if you want triggered assertions to print the assertion statement instead of aborting execution:</p> <pre>make_rtw OPTS="-DDOASSERTS -DPRINT_ASSERTS"</pre> <p><code>utAssert</code> is defined as <code>#define utAssert(exp) assert(exp)</code>.</p> <p>To customize assertion behavior, provide your own definition of <code>utAssert</code> in a handwritten header file that overrides the default <code>utAssert.h</code>. For details on how to include a</p>

To...	Select...
	customized header file in generated code, see “Configure Model for External Code Integration” on page 24-83. When running a model in accelerator mode, the Simulink engine calls back to itself to execute assertion blocks instead of using generated code. Thus, user-defined callbacks are still called when assertions fail.

For more information about the TLC debugging options, see Debugging on “Target Language Compiler”. Also, consider using the Model Advisor as a tool for troubleshooting model builds.

More About

- “Tunable Expression Limitations” on page 14-49
- “Configure Model for External Code Integration” on page 24-83
- “Target Language Compiler”

Target Configuration for Simulink Coder

- “Select a System Target File” on page 17-2
- “Specify Target Interfaces” on page 17-10
- “Configure Standard Math Library for Target System” on page 17-15
- “System Target Files and Code Generation Features” on page 17-18
- “Template Makefiles and Make Options” on page 17-26
- “Custom Targets” on page 17-32

Select a System Target File

The first step to configuring a model for code generation is to select and configure a system target file. When you select a system target file, other model configuration parameters change automatically to best serve requirements of the execution environment. For example:

- Code interface parameters
- Build process parameters, such as the toolchain or template makefile
- Target hardware parameters, such as word size and byte ordering

In this section...

“Target Platforms to Consider” on page 17-2

“System Target Files and Code Styles” on page 17-2

“Available System Target Files” on page 17-3

“System Target Files That Support Nonzero Start Time” on page 17-6

“Select a System Target File” on page 17-7

“Select a System Target File Programmatically” on page 17-8

Target Platforms to Consider

When you use Simulink software to create and simulate a model, and the code generator to produce code, you can consider three platforms:

- MATLAB Host — The host computer platform that runs MathWorks software during application development
- Production Target — The target hardware platform on which an application will be deployed when it is put into production
- Test Target — The platform on which an application under development is tested before deployment

For more information on configuring model code generation parameters for hardware platforms, see “Platform Options for Development and Deployment” on page 10-2.

System Target Files and Code Styles

The code generator produces two styles of code. One code style is suitable for rapid prototyping (and simulation by using code generation). The other style is suitable for

embedded applications. The following table maps system target files to corresponding code styles.

Code Styles Listed by System Target File

System Target File	Code Style (Using C or C++ Unless Noted)
Embedded Coder embedded real-time (ERT)	Embedded — Useful as a starting point when using generated C/C++ code in an embedded application (often referred to as a <i>production code target</i>).
Simulink Coder Generic real-time (GRT)	Rapid prototyping — Use as a starting point for creating a rapid prototyping target that does not use real-time operating system tasking primitives, and for verifying the generated code on your workstation.
Rapid simulation (RSim)	Rapid prototyping — Non-real-time simulation of your model on your workstation. Useful as a high-speed or batch simulation tool.
S-function	Rapid prototyping — Creates a C MEX S-function for simulation of your model within another Simulink model.
Simulink Desktop Real-Time	Rapid prototyping — Runs model in real time at interrupt level while your PC is running a Microsoft Windows environment in the background.
Simulink Real-Time	Rapid prototyping — Runs model in real time on target PC running the Simulink Real-Time kernel.

Third-party vendors supply additional system target files for the code generator. Generally, these can be classified as rapid prototyping system target files. For more information about third-party products, see the MathWorks Connections Program Web page: <http://www.mathworks.com/products/connections>.

Available System Target Files

The following table lists supported targets.

Note You can select from a range of system target files by using the System Target File Browser. This selection lets you experiment with configuration options and save your model with different configurations. However, you cannot build or generate code for non-GRT system target files unless you have the required license on your system (Embedded

Coder license for ERT system target files, Simulink Desktop Real-Time license for Simulink Desktop Real-Time system target files, and so on).

Selecting a system target file for your model selects either the toolchain approach or template makefile approach for build process control. For more information about these approaches, see “Configure Build Process” on page 25-13.

System Target Files Available from the System Target File Browser

System Target File	File Names	Reference
Embedded Coder (for PC or UNIX ^a platforms)	ert.tlc ert_shrplib.tlc	“Select an ERT System Target File” (Embedded Coder topic)
Create Visual C++ ^{®b} Solution File for Embedded Coder	ert.tlc (Requires RTW.MSVCBuild as TMF ^c)	“Select an ERT System Target File” (Embedded Coder topic)
Embedded Coder for AUTOSAR	autosar.tlc	“Generate Code That Is Compliant with the AUTOSAR Standard” (Embedded Coder topic)
Generic Real-Time (for PC or UNIX platforms)	grt.tlc	“System Target Files and Code Generation Features” on page 17-18
Create Visual C++ Solution File	grt.tlc (Requires RTW.MSVCBuild as TMF ^c)	“System Target Files and Code Generation Features” on page 17-18
Rapid Simulation (default for PC or UNIX platforms)	rsim.tlc	“Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” on page 22-2
Rapid Simulation for LCC compiler	rsim.tlc	“Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” on page 22-2
Rapid Simulation for UNIX platforms	rsim.tlc	“Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” on page 22-2
Rapid Simulation for Visual C++ compiler	rsim.tlc	“Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” on page 22-2

System Target File	File Names	Reference
S-Function for PC or UNIX platforms	rtwsfcn.tlc	“Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target” on page 22-33
S-Function for LCC	rtwsfcn.tlc	“Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target” on page 22-33
S-Function for UNIX platforms	rtwsfcn.tlc	“Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target” on page 22-33
S-Function for Visual C++ compiler	rtwsfcn.tlc	“Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target” on page 22-33
ASAM-ASAP2 Data Definition	asap2.tlc	“Export ASAP2 File for Data Measurement and Calibration” on page 29-2
Simulink Desktop Real-Time	sldrt.tlc sldrtert.tlc	“Set External Mode Code Generation Parameters” (Simulink Desktop Real-Time topic)
Simulink Real-Time	slrt.tlc slrtert.tlc	“Code Generation Configuration Parameters” (Simulink Real-Time topic)
IDE Link capability	idelink_grt.tlc idelink_ert.tlc	Embedded IDE/target topics such as “Model Setup” (Embedded Coder topic)

- a. UNIX is a registered trademark of The Open Group in the United States and other countries.
- b. Visual C++ is a registered trademark of Microsoft Corporation.
- c. Set RTW.MSVCCBuild in the “Template makefile” field. This creates and builds Visual C++ Solution (.sln) file with Debug configuration.

System Target Files That Support Nonzero Start Time

When you try to build models with a nonzero start time, if the selected system target file does not support a nonzero start time, the code generator does not produce code and displays an error message. The Rapid Simulation (RSim) system target file supports a nonzero start time when **Configuration Parameters > RSim Target > Solver selection** is set to Use Simulink solver module. Other system target files do not support a nonzero start time.

Select a System Target File

Use the **Browse** button on the **Code Generation** pane to open the System Target File Browser and select a system target file. Your selection appears in the **System target file** field (*target.tlc*).

If you use a target configuration that does not appear in the System Target File Browser, enter the name of your system target file in the **System target file** field. Click **Apply** or **OK** to configure for that target.

You also can select a system target file programmatically from MATLAB code, as described in “Select a System Target File Programmatically” on page 17-8.

After selecting a system target file, you can modify model configuration parameter settings. Selecting a system target file for your model selects either the toolchain approach or template makefile approach for build process control. For more information about these approaches, see “Configure Build Process” on page 25-13.

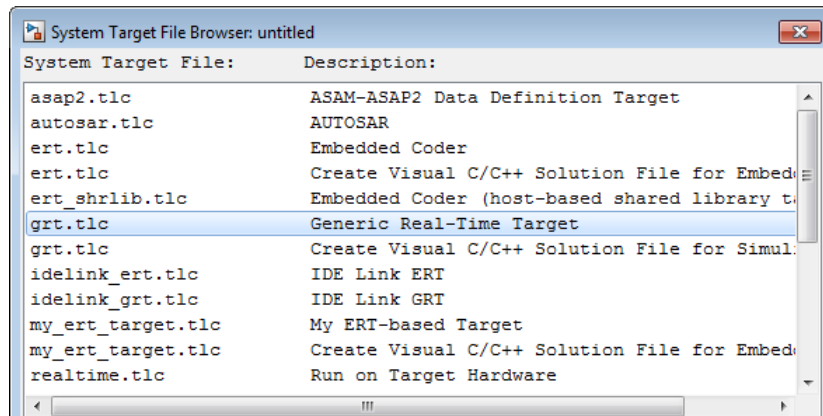
If you want to switch between different targets in a single workflow for different code generation purposes (for example, rapid prototyping versus product code deployment), set up different configuration sets for the same model and switch the active configuration set for the current operation. For more information on how to set up configuration sets and change the active configuration set, see “Manage a Configuration Set” in the Simulink documentation.

To select a system target file using the System Target File Browser,

- 1 Open the **Code Generation** pane of the Configuration Parameters dialog box.
- 2 Click the **Browse** button next to the **System target file** field. This opens the System Target File Browser. The browser displays a list of currently available system target files, including customizations. When you select a system target file, the code generator automatically chooses the system target file, toolchain or template makefile, and/or **make** command for that configuration.

The next step shows the System Target File Browser with the generic real-time target selected.

- 3 Click the desired entry in the list of available configurations. The background of the list box turns yellow to indicate an unapplied choice has been made. To apply it, click **Apply** or **OK**.



System Target File Browser

When you choose a system target file, the code generator automatically chooses the system target file, toolchain or template makefile, and/or `make` command for that configuration, and displays them in the **System target file** field. The description of the system target file from the browser is placed below its name in the **Code Generation** pane.

Select a System Target File Programmatically

Simulink models store model-wide parameters and target-specific data in *configuration sets*. Every configuration set contains a component that defines the structure of a particular system target file and the current values of relevant options. Some of this information is loaded from the system target file that you specify. You can configure models to generate alternative code by copying and modifying old or adding new configuration sets and browsing to select a new system target file. Subsequently, you can interactively select an active configuration from among these sets (only one configuration set can be active at a given time).

Scripts that automate target selection need to emulate this process.

To program target selection

- 1 Obtain a handle to the active configuration set with a call to the `getActiveConfigSet` function.

- 2 Define character vector variables that correspond to the required system target file, toolchain or template makefile, and/or `make` command settings. For example, for the ERT target, you would define variables for the character vectors `'ert.tlc'`, `'ert_default_tmf'`, and `'make_rtw'`.
- 3 Select the system target file with a call to the `switchTarget` function. In the function call, specify the handle for the active configuration set and the system target file.
- 4 Set the `TemplateMakefile` and `MakeCommand` configuration parameters to the corresponding variables created in step 2.

For example:

```
cs = getActiveConfigSet(model);
stf = 'ert.tlc';
tmf = 'ert_default_tmf';
mc = 'make_rtw';
switchTarget(cs,stf,[]);
set_param(cs,'TemplateMakefile',tmf);
set_param(cs,'MakeCommand',mc);
```

For more information about selecting system target files programmatically, see `switchTarget`.

See Also

`getActiveConfigSet` | `switchTarget`

More About

- “System Target Files and Code Generation Features” on page 17-18
- “Specify Target Interfaces” on page 17-10

Specify Target Interfaces

Use interface model configuration parameters to control which libraries you want to use when generating code, whether to include support for an API in generated code, and other interface options.

To...	Select or Enter...
Specify the standard math library to use when generating code	<p>Select C89/C90 (ANSI), C99 (ISO), or C++03 (ISO) for the Standard math library parameter on the All Parameters tab.</p> <p>Selecting C89/C90 (ANSI) provides the ANSI^a C set of library functions. For example, selecting C89/C90 (ANSI) results in generated code that calls <code>sin()</code> whether the input argument is double precision or single precision. However, if you select C99 (ISO), the generated code calls the function <code>sinf()</code> when the input argument is single precision. If your compiler supports the ISO^{®b} C math extensions, selecting the ISO C library can result in more efficient code.</p> <p>For more information, see “Standard math library”</p>
Specify an application-specific library to use when generating code	<p>If you need to generate application-specific C or C++ code for math functions or operations, select a value for Code replacement library. Otherwise, specify None.</p> <p>For more information about code replacement libraries, see “Choose a Code Replacement Library” on page 21-8 and “Code replacement library”</p>
Direct where the code generator should place fixed-point and other utility code	<p>Select Auto or Shared location for Shared code placement. The shared location directs code for utilities to be placed within the <code>slprj</code> folder in your working folder, which is used for building model reference targets. If you select Auto,</p> <ul style="list-style-type: none"> • When the model contains Model blocks, places utility code within the <code>slprj/target/_sharedutils</code> folder. • When the model does not contain Model blocks, places utility code in the build folder (generally, in <code>model.c</code> or <code>model.cpp</code>).
Specify text to be added to the variable names used	<p>Enter a prefix or suffix, such as <code>rt_</code> or <code>_rt</code>, for the MAT-file variable name modifier parameter on the All Parameters tab.</p>

To...	Select or Enter...
when logging data to MAT-files, to distinguish logging data from code generation and simulation applications	The code generator prefixes or appends the text to the variable names for system outputs, states, and simulation time specified in the Data Import/Export pane. See “Log Program Execution Results” on page 27-2 for information on MAT-file data logging.
Specify data exchange APIs to be included in generated code	Select one or more C API options, the ASAP2 interface option, or the External mode option. When you select External mode , other options appear. The data exchange APIs are mutually independent and can be selected in combinations, for example, C API and external mode. For more information on working with these interfaces, see “Exchange Data Between Generated and External Code Using C API” on page 28-2, “Export ASAP2 File for Data Measurement and Calibration” on page 29-2, and “What You Can Do With a Host/Target Communication Channel” on page 26-2.

- a. ANSI is a registered trademark of the American National Standards Institute, Inc.
- b. ISO is a registered trademark of the International Organization for Standardization.

Note: Before setting **Standard math library** or **Code replacement library**, verify that your compiler supports the library you want to use. If you select a parameter value that your toolchain does not support, compiler errors can occur. For example, if you select standard math library **C99 (ISO)** and your compiler does not support the ISO C math extensions, compile-time errors might occur.

When the Embedded Coder product is installed on your system, the **Code Generation > Interface** pane expands to include several additional options. For descriptions of **Code Generation > Interface** pane parameters, see “Model Configuration Parameters: Code Generation Interface”.

Interface Dependencies

Several interface parameters have dependencies on settings of other parameters. The following table summarizes the dependencies.

Parameter	Dependencies?	Dependency Details
Standard math library	Yes	Available values depend on Language selection.
Code replacement library	Yes	Available values depend on product licensing and other parameters. For more information, see “Code replacement library”.
Shared code placement	No	
Support: floating-point numbers (ERT targets only)	No	
Support: non-finite numbers	Yes (ERT targets) No (GRT targets)	For ERT targets, enabled by Support floating-point numbers
Support: complex numbers (ERT targets only)	No	
Support: absolute time (ERT targets only)	No	
Support: continuous time (ERT targets only)	Yes	Requires that you disable Remove error status field in real-time model data structure .
Support: non-inlined S-functions (ERT targets only)	Yes	Requires that you enable Support floating-point numbers and Support non-finite numbers
Classic call interface	Yes	Requires that you disable Single output/update function . For ERT targets, requires that you enable Support floating-point numbers .
Single output/update function	Yes	Disable for Classic call interface
Terminate function required (ERT targets only)	Yes	
Code interface packaging	Yes	Available values depend on Language selection.
Multi-instance code error diagnostic	Yes	Set Code interface packaging to Reusable function or C++ class

Parameter	Dependencies?	Dependency Details
Pass root-level I/O as (ERT targets only)	Yes	Set Code interface packaging to Reusable function
Use dynamic memory allocation for model initialization (ERT targets only)	Yes	Set Code interface packaging to Reusable function
MAT-file logging	Yes	For GRT system target files, requires that you enable Support non-finite numbers ; for ERT system target files, requires that you enable Support floating-point numbers , Support non-finite numbers , and Terminate function required
MAT-file file variable name modifier	Yes	Enabled by MAT-file logging
Remove error status field in real-time model data structure (ERT targets only)	Yes	Requires that you disable Support: continuous time .
Generate C API for: signals	No	
Generate C API for: parameters	No	
Generate C API for: states	No	
Generate C API for: root-level I/O	No	
ASAP2 interface	No	
External mode	No	
Transport layer	Yes	Enable External mode
MEX-file arguments	Yes	Enable External mode
Static memory allocation	Yes	Enable External mode
Static memory buffer size	Yes	Enable Static memory allocation

More About

- “System Target Files and Code Generation Features” on page 17-18

- “Select a System Target File” on page 17-2
- “Configure Standard Math Library for Target System” on page 17-15
- “Replace Code Generated from Simulink Models” on page 21-10

Configure Standard Math Library for Target System

Specify standard library extensions that the code generator uses for math operations.

When you generate code for a new model or with a new configuration set object, the code generator uses the ISO@/IEC 9899:1999 C (C99 (ISO)) library by default. For preexisting models and configuration set objects, the code generator uses the library specified by the **Standard math library** parameter.

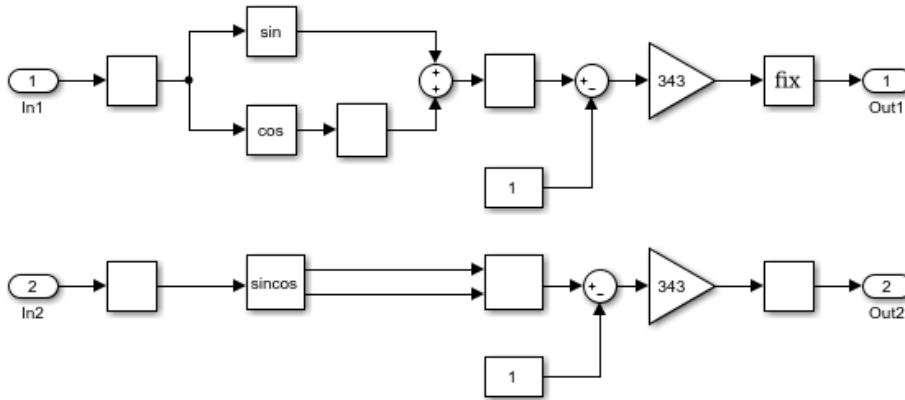
If your compiler supports the ISO@/IEC 9899:1990 (C89/C90 (ANSI)) or ISO/IEC 14882:2003(C++03 (ISO)) math library extensions, you can change the standard math library setting. The C++03 (ISO) library is an option when you select C++ for the programming language.

The C99 library leverages the performance that a compiler offers over standard ANSI C. When using the C99 library, the code generator produces calls to ISO C functions when possible. For example, the generated code calls the function `sqrtf()`, which operates on single-precision data, instead of `sqrt()`.

To change the library setting, use the **Configuration Parameters>All Parameters>Standard math library** parameter. The command-line equivalent is `TargetLangStandard`.

Generate and Inspect ANSI C Code

1. Open the example model `rtwdemo_clibsup`.



Use C89/C90 (ANSI) Math Library (double-click)

Use C99 (ISO) Math Library (double-click)

Use C++03 (ISO) Math Library (double-click)

View Interface Configuration (double-click)

By default, Simulink Coder and Embedded Coder generate C89/C90 (ANSI) code, and both support C99 (ISO), and C++03 (ISO) (available when TargetLang=C++) math library extensions. This model shows math library extensions provided by the Code Generation Interface options "Standard math library" and "Code replacement library".

For example, if your compiler supports the C99 (ISO) (ISO/IEC 9899:1999) math library, calls to the C99 (ISO) functions (for example, `sqrtf()` instead of `sqrt()` for single-precision data) are generated whenever possible. This option leverages the performance your target compiler offers over standard C89/C90 (ANSI).

Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

Copyright 1994-2015 The MathWorks, Inc.

2. Generate code.

```
### Starting build procedure for model: rtwdemo_clibsup
### Successful completion of code generation for model: rtwdemo_clibsup
```

3. Examine the code in the generated file `rtwdemo_clibsup.c`. Note that the code calls the `sqrt` function.

```
if (rtb_Abs2 < 0.0F) {
    rtb_Abs2 = -(real32_T)sqrt((real32_T)fabs(rtb_Abs2));
} else {
    rtb_Abs2 = (real32_T)sqrt(rtb_Abs2);
}
```


Generate and Inspect ISO C Code

1. Change the setting of **All Parameters>Standard math library** to **C99 (ISO)**. Alternatively, at the command line, set `TargetLangStandard` to **C99 (ISO)**.

2. Regenerate the code.

```
### Starting build procedure for model: rtwdemo_clibsup  
### Successful completion of code generation for model: rtwdemo_clibsup
```

3. Reexamine the code in the generated file `rtwdemo_clibsup.c`. Now the generated code calls the function `sqrtf` instead of `sqrt`.

```
if (rtb_Abs2 < 0.0F) {  
    rtb_Abs2 = -sqrtf(fabsf(rtb_Abs2));  
} else {  
    rtb_Abs2 = sqrtf(rtb_Abs2);  
}
```

Related Information

- “Standard math library”
- “Select a System Target File”

More About

- “Select a System Target File” on page 17-2
- “Replace Code Generated from Simulink Models” on page 21-10

System Target Files and Code Generation Features

A system target file (such as `grt.tlc`) defines a *target* (such as the generic real-time target). The code generator uses the system target file to produce code intended for execution on a certain hardware or operating system platform. The system target file invokes other target-specific files.

Different types of system target files support a variety of generated code features. In the system target file, the value of the `CodeFormat` TLC variable and corresponding `rtwgensettings.DerivedFrom` field value identify the target type and generated code features. These selections control decisions made at several points in the code generation process. These decisions include whether and how certain data structures are generated (for example, `SimStruct` or `rtModel`), whether or not static or dynamic memory allocation code is generated, and the calling interface used for generated model functions. For custom system target file development, follow these guidelines:

- If the system target file does not include a value for the `CodeFormat` TLC variable, the default value is `RealTime` for generic real-time target (GRT). The corresponding `rtwgensettings.DerivedFrom` field value is `grt.tlc` (default value).
- If you are developing a custom target and you have a license for Embedded Coder software, consider setting the `CodeFormat` TLC variable value to `Embedded-C` for embedded real-time target (ERT). The corresponding `rtwgensettings.DerivedFrom` field value is `ert.tlc`. The ERT target supports more generated code features than the GRT target.

The following example shows how the value for the `CodeFormat` TLC variable and corresponding `rtwgensettings.DerivedFrom` field value are set in `ert.tlc`.

```
%assign CodeFormat = "Embedded-C"  
  
/%  
  BEGIN_RTW_OPTIONS  
  rtwgensettings.DerivedFrom = 'ert.tlc';  
  END_RTW_OPTIONS  
%/
```

Warning You must use the value for the `CodeFormat` TLC variable with its corresponding `rtwgensettings.DerivedFrom` field value to generated code for the model. If none are explicitly selected, the default values correspond. For more information, see “System Target File Structure” on page 37-30.

For a description of the optimized call interface generated by default for both the GRT and ERT targets, see “Entry-Point Functions and Scheduling” on page 15-2.

The real-time model data structure (`rtModel`) encapsulates model-specific information in a much more compact form than the `SimStruct`. Many efficiencies related to generated code depend on generation of `rtModel` rather than `SimStruct`, including:

- Integer absolute and elapsed timing services
- Independent timers for asynchronous tasks
- Generation of improved C API code for signal, state, and parameter monitoring
- Pruning the data structure to minimize its size (ERT-derived targets only)

For a description of the `rtModel` data structure, see “Use the Real-Time Model Data Structure” on page 14-19.

The following topics provide more information about generated code features:

In this section...
“Compare Generated Code Features by Target” on page 17-19
“Compare Generated Code Features by Product” on page 17-22

Compare Generated Code Features by Target

The code generator supports a variety of generated code features for different types of targets. In the system target file for each target, the set of features is identified with the value of the `CodeFormat` TLC variable.

The following table summarizes how different targets support applications:

Application	Targets
Fixed- or variable-step acceleration	RSIM, S-Function, Model Reference
Fixed-step real-time deployment	GRT, ERT, Simulink Real-Time, Simulink Desktop Real-Time, ...

The following table summarizes the various options available for each **System target file** selection, with the exceptions noted.

Features Supported by System Target File (STF) Targets

	System Target File (STF) Targets							
Feature	grt.tlc ¹	ert.tlc ¹	ert_shrplib.tlc ¹	rtwscfn.tlc ¹	rsim.tlc ¹	sldrt.tlc ¹	slrt.tlc ¹	Other ¹
Static memory allocation	X	X				X	X	X
Dynamic memory allocation	X ^{4, 5}	X ^{4, 5}		X	X		X	
Continuous time	X	X		X	X	X	X	
C/C++ MEX S-functions (noninlined)	X	X		X	X	X	X	
S-function (inlined)	X	X		X	X	X	X	X
Minimize RAM/ROM usage		X					X ²	X
Supports external mode	X	X			X	X	X	
Rapid prototyping	X					X	X	X
Production code		X					X ²	X ³
Batch parameter tuning and Monte Carlo methods			X		X			
System-level Simulator			X					

Feature	System Target File (STF) Targets							
	grt.tlc ¹	ert.tlc ¹	ert_shrplib.tlc ¹	rtwscfn.tlc ¹	rsim.tlc ¹	sldrt.tlc ¹	slrt.tlc ¹	Other ¹
Executes in hard real time	X ³	X ³				X	X	X ⁵
Non-real-time executable included	X	X			X			
Multiple instances of model	X ^{4, 5}	X ^{4, 5}		X ⁴			X ^{4, 5}	X ^{4, 5}
Supports variable-step solvers				X	X			
Supports SIL/PIL		X						X

¹ System Target Files: **grt.tlc** - Generic Real-Time Target, **ert.tlc** - Embedded Coder, **ert_shrplib.tlc** - Embedded Coder (develop platform shared library), **rtwscfn.tlc** - S-Function, **rsim.tlc** - Rapid Simulation, **sldrt.tlc** - Simulink Desktop Real-Time, **slrt.tlc** - Simulink Real-Time, and **Other** - The Embedded Targets capabilities in Simulink Coder support other targets.

²Does not apply to GRT based system target files. Applies only to an ERT based system target files.

³The default GRT and ERT `rt_main` files emulate execution of hard real time, and when explicitly connected to a real-time clock execute in hard real time.

⁴You can generate code for multiple instances of a Stateflow chart or subsystem containing a chart, except when the chart contains exported graphical functions or the Stateflow model contains machine parented events.

⁵You must select the value **Reusable function** for **Code interface packaging** in the **Code Generation > Interface** pane of the Configuration Parameters dialog box.

Compare Generated Code Features by Product

The code generation process for real-time system target files (such as GRT) provides many embedded code optimizations. GRT and ERT system target files have many common features. But, selecting an ERT based system target file offers more extensive features. The availability of features are determined by code generation product as selected by the system target file. The following table compares code features available with Simulink Coder and features available with Embedded Coder.

Compare Code Generation Features for Simulink Coder Versus Embedded Coder

Feature	Simulink Coder	Embedded Coder
rtModel data structure	<ul style="list-style-type: none"> Full rtModel structure generated GRT variable declaration: <code>rtModel_model model_M;</code> 	<ul style="list-style-type: none"> rtModel is optimized for the model Optional suppression of error status field and data logging fields ERT variable declaration: <code>RT_MODEL_model model_M;</code>
Custom storage classes (CSCs)	Code generation ignores CSCs; objects are assigned a CSC default to Auto storage class	Code generation with CSCs is supported
HTML code generation report	Basic HTML code generation report	Enhanced report with additional detail and hyperlinks to the model
Symbol formatting	Symbols (for signals, parameters and so on) are generated in accordance with hard-coded default	Detailed control over generated symbols.
User-defined maximum identifier length for generated symbols	Supported	Supported
Generation of terminate function	Generated	Option to suppress terminate function
Combined output/update function	Separate output/update functions are generated	Option to generate combined output/update function
Optimized data initialization	Not available	Options to suppress generation of unnecessary initialization code for

Feature	Simulink Coder	Embedded Coder
		zero-valued memory, I/O ports, and so on
Comments generation	Basic options to include or suppress comment generation	Options to include Simulink block descriptions, Stateflow object descriptions, and Simulink data object descriptions in comments
Module Packaging Features (MPF)	Not supported	Extensive code customization features (see Embedded Coder topics such as “What Are User-Defined Data Types?” and “Custom Storage Classes”).
Target-optimized data types header file	Requires full <code>tmwtypes.h</code> header file	Generates optimized <code>rtwtypes.h</code> header file, including definitions required by the target
User-defined types	User-defined types default to base types in code generation	User defined data type aliases are supported in code generation
Rate grouping	Not supported	Supported
Auto-generation of main program module	Not supported; static main program module is provided.	Automated and customizable generation of main program module is supported (static main program also available)
Reusable (multi-instance) code generation	Option to generate reusable code with dynamic memory allocation	Option to generate reusable code with static or dynamic memory allocation
Software constraint options	Support for floating point, complex, and nonfinite numbers is enabled	Options to enable or disable support for floating-point, complex, and nonfinite numbers
Application life span	Defaults to <code>inf</code>	User-specified; determines most efficient word size for integer timers
Software-in-the-loop (SIL) testing	Model reference simulation target can be used for SIL testing	Additional SIL testing support by using auto-generation of SIL block
ANSI ^a -C/C++ code generation	Supported	Supported

Feature	Simulink Coder	Embedded Coder
ISO ^b -C/C++ code generation	Supported	Supported
GNU ^c -C/C++ code generation	Supported	Supported
Generate scalar inlined parameters as #DEFINE statements	Not supported	Supported
MAT-file variable name modifier	Supported	Supported
Data exchange: C API, ASAP2, external mode	Supported	Supported

- a. ANSI is a registered trademark of the American National Standards Institute, Inc.
b. ISO is a registered trademark of the International Organization for Standardization.
c. GNU is a registered trademark of the Free Software Foundation.

More About

- “Select a System Target File” on page 17-2
- “Code Appearance”
- “System Target File Structure” on page 37-30
- “Entry-Point Functions and Scheduling” on page 15-2
- “Use the Real-Time Model Data Structure” on page 14-19
- “What Are User-Defined Data Types?”
- “Custom Storage Classes”
- “S-Functions and Code Generation” on page 24-99
- “Inlining S-Functions”
- **Code interface packaging**
- “Use GRT with Reusable Function Packaging to Combine Models” on page 9-4
- “Embedded Coder Product Description”
- “Code Appearance”
- “Configure Model for Code Generation Objectives Using Code Generation Advisor”
- “Select an ERT System Target File”

External Websites

- `rt_malloc_main.c`
- `matlabroot/rtw/c/src/common`

Template Makefiles and Make Options

The code generator includes a set of built-in template makefiles that are designed to build programs for specific targets.

In this section...

“Types of Template Makefiles” on page 17-26

“Specify Template Makefile Options” on page 17-27

“Template Makefiles for UNIX Platforms” on page 17-27

“Template Makefiles for the Microsoft Visual C++ Compiler” on page 17-28

“Template Makefiles for the LCC Compiler” on page 17-30

Types of Template Makefiles

There are two types of template makefiles:

- *Compiler-specific* template makefiles are designed for use with a particular compiler or development system.

By convention, compiler-specific template makefiles are named according to the system target file and compiler (or development system). For example, `grt_vc.tmf` is the template makefile for building a generic real-time program under the Visual C++ compiler; `ert_lcc.tmf` is the template makefile for building an Embedded Coder program under the Lcc compiler.

- *Default* template makefiles make your model designs more portable, by choosing the compiler-specific makefile and compiler for your installation. “Select and Configure Compiler or IDE” on page 25-3 describes the operation of default template makefiles in detail.

Default template makefiles are named *target_default_tmf*. They are MATLAB language files that, when run, select the TMF for the specified target configuration. For example, `grt_default_tmf` is the default template makefile for building a generic real-time program; `ert_default_tmf` is the default template makefile for building an Embedded Coder program.

Complete details on the structure of template makefiles are provided in “Customize Template Makefiles” on page 37-62. This section describes compiler-specific template makefiles and common options you can use with each.

Specify Template Makefile Options

You can specify template makefile options by using the **Make command** field in the **Code Generation** pane of the Configuration Parameters dialog box. Append the options after `make_rtw` (or other `make` command), as in the following example:

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

The syntax for `make` command options differs slightly for different compilers.

Note: To control compiler optimizations for a makefile build at the Simulink GUI level, use the **Compiler optimization level** parameter on the **All Parameters** tab of the Configuration Parameters dialog box. The **Compiler optimization level** parameter provides

- Target-independent values **Optimizations on (faster runs)** and **Optimizations off (faster builds)**, which allow you to easily toggle compiler optimizations on and off during code development
- The value **Custom** for entering custom compiler optimization flags at Simulink GUI level (rather than at other levels of the build process)

If you specify compiler options for your makefile build using `OPT_OPTS`, `MEX_OPTS` (except `MEX_OPTS=" -v "`), or `MEX_OPT_FILE`, the value of **Compiler optimization level** is ignored and a warning is issued about the ignored parameter.

Template Makefiles for UNIX Platforms

The template makefiles for UNIX platforms are designed to be used with the Free Software Foundation's GNU Make. These makefile are set up to conform to the guidelines specified in the IEEE^{®4} Std 1003.2-1992 (POSIX) standard.

- `ert_unix.tmf`
- `grt_unix.tmf`
- `rsim_unix.tmf`
- `rtwsfcn_unix.tmf`

4. IEEE is a registered trademark of The Institute of Electrical and Electronics Engineers, Inc.

You can supply options to makefiles using the **Make command** field on the **Code Generation** pane of the Configuration Parameters dialog box. Options specified in **Make command** are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. The following options can be used to modify the behavior of the build:

- **OPTS** — User-specific options, for example,

```
OPTS=" -DMYDEFINE=1 "
```
- **OPT_OPTS**— Optimization options. Default is `-O`. To enable debugging specify as `OPT_OPTS=-g`. Because of optimization problems in `IBM_RS`, the default is no optimization.
- **CPP_OPTS** — C++ compiler options.
- **USER_SRCS** — Additional user sources, such as files used by S-functions.
- **USER_INCLUDES** — Additional include paths, for example,

```
USER_INCLUDES=" -Iwhere-ever -Iwhere-ever2 "
```
- **DEBUG_BUILD** — Add debug information to generated code, for example,

```
DEBUG_BUILD=1
```

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for the Microsoft Visual C++ Compiler

- “Visual C++ Executable Build” on page 17-28
- “Visual C++ Code Generation Only” on page 17-29

Visual C++ Executable Build

To build an executable using the Visual C++ compiler within the build process, use one of the `target_vc.tmf` template makefiles:

- `ert_vc.tmf`
- `grt_vc.tmf`
- `rsim_vc.tmf`
- `rtwsfcn_vc.tmf`

You can supply options to makefiles using the **Make command** field on the **Code Generation** pane of the Configuration Parameters dialog box. Options specified in **Make command** are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. The following options can be used to modify the behavior of the build:

- `OPT_OPTS` — Optimization option. Default is `-O2`. To enable debugging specify as `OPT_OPTS=-Zi`.
- `OPTS` — User-specific options.
- `CPP_OPTS` — C++ compiler options.
- `USER_SRCS` — Additional user sources, such as files used by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,
`USER_INCLUDES=" -Iwhere-ever -Iwhere-ever2"`
- `DEBUG_BUILD` — Add debug information to generated code, for example,
`DEBUG_BUILD=1`

These options are also documented in the comments at the head of the respective template makefiles.

Visual C++ Code Generation Only

To create a Visual C++ project makefile (*model.mak*) without building an executable, use one of the *target_msvc.tmf* template makefiles:

- `ert_msvc.tmf`
- `grt_msvc.tmf`

These template makefiles are designed to be used with `nmake`, which is bundled with the Visual C++ compiler.

You can supply options to makefiles using the **Make command** field on the **Code Generation** pane of the Configuration Parameters dialog box. Options specified in **Make command** are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. The following options can be used to modify the behavior of the build:

- `OPTS` — User-specific options, for example,
`OPTS="/D MYDEFINE=1"`

- `USER_SRCS` — Additional user sources, such as files used by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,
`USER_INCLUDES=" -Iwhere-ever -Iwhere-ever2"`
- `DEBUG_BUILD` — Add debug information to generated code, for example,
`DEBUG_BUILD=1`

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for the LCC Compiler

The code generator provides template makefiles to create an executable for the Windows platform using Lcc compiler Version 2.4 and GNU Make (gmake).

- `ert_lcc.tmf`
- `grt_lcc.tmf`
- `rsim_lcc.tmf`
- `rtwsfcn_lcc.tmf`

You can supply options to makefiles using the **Make command** field on the **Code Generation** pane of the Configuration Parameters dialog box. Options specified in **Make command** are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. The following options can be used to modify the behavior of the build:

- `OPTS` — User-specific options, for example,
`OPTS=" -DMYDEFINE=1 "`
- `OPT_OPTS` — Optimization options. Default is no options. To enable debugging, specify `-g4`:
`OPT_OPTS=" -g4 "`
- `CPP_OPTS` — C++ compiler options.
- `USER_SRCS` — Additional user sources, such as files used by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,
`USER_INCLUDES=" -Iwhere-ever -Iwhere-ever2"`

For Lcc, use / as file separator before the filename instead of \, for example, `d:\work\proj1/myfile.c`.

- `DEBUG_BUILD` — Add debug information to generated code, for example,

```
DEBUG_BUILD=1
```

These options are also documented in the comments at the head of the respective template makefiles.

More About

- “Select a System Target File” on page 17-2
- “Custom Targets” on page 17-32

Custom Targets

You can create your own system target files to build custom targets that interface with external code or operating environments.

For more information on how to make custom targets appear in the System Target File Browser and display relevant controls, see “About Embedded Target Development” on page 37-2 and the topics it references.

More About

- “Select a System Target File” on page 17-2
- “Specify Target Interfaces” on page 17-10
- “System Target Files and Code Generation Features” on page 17-18

Internationalization Support in Simulink Coder

Internationalization and Code Generation

Internationalization support in software development tooling is vital for enabling efficient globalization. If there is a remote possibility that you will collaborate with others across locales, consider internationalization from project inception. Internationalization can prevent rework or having to develop a new model design. The relevant requirement concerns locale settings.

In this section...

“Locale Settings” on page 18-2

“Prepare to Generate Code for Mixed Languages and Locales” on page 18-2

“XML Escape Sequence Replacements” on page 18-3

“Character Set Limitations” on page 18-3

Locale Settings

A locale setting defines the language (character set encoding) for the user interface and the display formats on a computer for information such as time, date, and currency. The encoding dictates the number of characters a locale can render. For example, the US-ASCII coded character set (codeset) defines 128 characters. A Unicode[®] codeset, such as UTF-8, defines more than 1,100,000 characters.

For code generation, the locale setting determines the character set encoding of generated file content. To avoid garbled text or incorrectly displayed characters, the locale setting for your MATLAB session must be compatible with the setting for your compiler and operating system. For information on finding and changing the operating system setting, see “Internationalization” in the MATLAB documentation or the operating system documentation.

To check a model for characters that cannot be represented in the locale setting of current MATLAB session, use the Simulink Model Advisor check “Check model for foreign characters”.

Prepare to Generate Code for Mixed Languages and Locales

To prepare to generate code for a model, identify:

- The operating system locale.

- The locale of the MATLAB session.
- Code generation requirements for Target Language Compiler files.

XML Escape Sequence Replacements

The code generator replaces characters that are not represented in the character set encoding of a model with XML escape sequences. Escape sequence replacements occur for block, signal, and Stateflow object names that appear in:

- Generated code comments
- Code generation reports
- Block paths logged to MAT-files
- Block paths logged to C API files *model_capi.c* (or *.cpp*) and *model_capi.h*

Character Set Limitations

Target Language Compiler files support user default encoding only. To produce international custom generated code that is portable, use the 7-bit ASCII character set.

More About

- “Locale Settings for MATLAB Process”

Source Code Generation in Simulink Coder

- “Initiate Code Generation” on page 19-2
- “Model and Test Environment” on page 19-3
- “Configure Model and Generate Code” on page 19-14
- “Configure Data Interface” on page 19-21
- “Call External C Functions” on page 19-30
- “Reload Generated Code” on page 19-37
- “Generated Source Files and File Dependencies” on page 19-38
- “Files and Folders Created During Build Process” on page 19-57
- “Enable Build Process When Folder Names Have Spaces” on page 19-65
- “How Code Is Generated from a Model” on page 19-71
- “Code Generation of Matrices and Arrays” on page 19-73
- “Generated Code Considerations” on page 19-77
- “About Shared Utility Code” on page 19-78
- “Controlling Shared Utility Code Placement” on page 19-79
- “rtwtypes.h and Shared Utility Code” on page 19-81
- “Incremental Shared Utility Code Generation and Compilation” on page 19-82
- “Shared Utility Checksum” on page 19-83
- “Shared Fixed-Point Utility Functions” on page 19-88
- “Share User-Defined Data Types Across Models” on page 19-90
- “Cross-Release Shared Code Reuse” on page 19-95
- “Cross-Release Code Integration” on page 19-98
- “Generate Code Using Simulink® Coder™” on page 19-106

Initiate Code Generation

You can generate code for your model with or without the compilation, linking, and other processing that occurs as part of a full model build. To generate code without initiating a full model build, select the **Generate code only** option on the **Code Generation** pane of the Configuration Parameters dialog box and press **Ctrl+B**. To initiate a full model build, clear the **Generate code only** option and press **Ctrl+B**, or use a method described in “Initiate the Build Process” on page 25-35.

When your default folder is set to the root folder of a drive, such as **C:**, the code generator cannot generate code for your model.

Model and Test Environment

In this section...

- “About This Example” on page 19-3
- “Functional Design of the Model” on page 19-4
- “View the Top Model” on page 19-4
- “View the Subsystems” on page 19-5
- “Simulation Test Environment” on page 19-6
- “Run Simulation Tests” on page 19-11
- “Key Points” on page 19-12
- “Learn More” on page 19-13

About This Example

Learning Objectives

- Learn about the functional behavior of the example model.
- Learn about the role of the example test harness and its components.
- Run simulation tests on a model.

Prerequisites

- Ability to open and modify Simulink models and subsystems.
- Understand subsystems and how to view subsystem details.
- Understand referenced models and how to view referenced model details.
- Ability to set model configuration parameters.

Required Files

Before you use each example model file, place a copy in a writable location and add it to your MATLAB path.

- `rtwdemo_throttlecntrl` model file
- `rtwdemo_throttlecntrl_testharness` model file

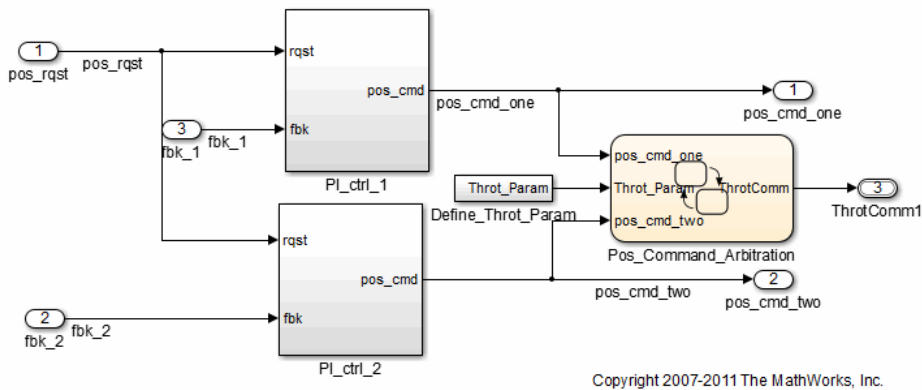
Functional Design of the Model

This example uses a simple, but functionally complete, example model of a throttle controller. The model features redundant control algorithms. The model highlights a standard model structure and a set of basic blocks in algorithm design.

View the Top Model

Open `rtwdemo_throttlecntrl` and save a copy as `throttlecntrl` in a writable location on your MATLAB path.

Note: This model uses Stateflow software.



The top level of the model consists of the following elements:

Subsystems	PI_ctrl_1 PI_ctrl_2 Define_Throt_Param Pos_Command_Arbitration
Top-level input	pos_rqst fbk_1 fbk_2

Top-level output	pos_cmd_one pos_cmd_two ThrotComm1
Signal routing	
<i>Omit</i> blocks that change the value of a signal, such as Sum and Integrator	

The layout uses a basic architectural style for models:

- Separation of calculations from signal routing (lines and buses)
- Partitioning into subsystems

You can apply this style to a wide range of models.

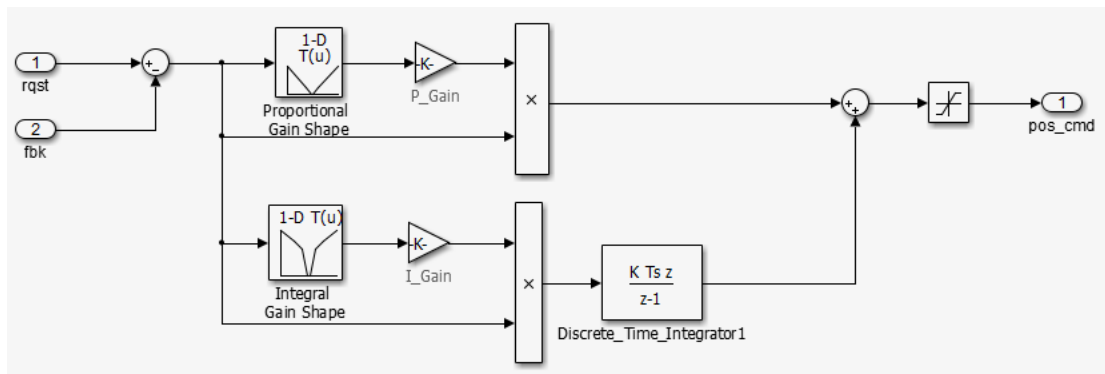
View the Subsystems

Explore two of the subsystems in the top model.

- 1 If not already open, open `throttlecntrl`.

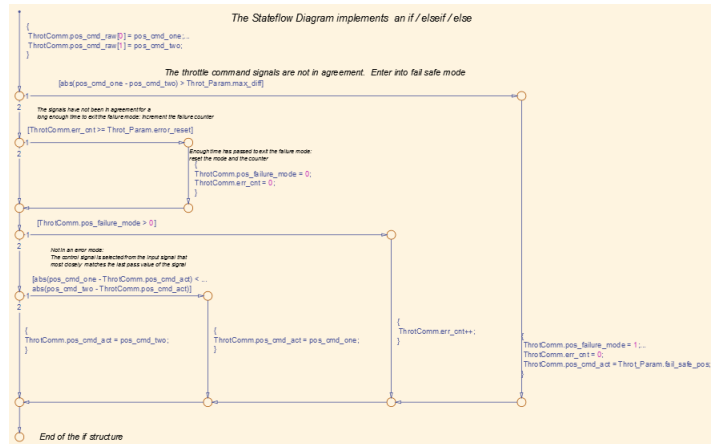
Two subsystems in the top model represent proportional-integral (PI) controllers, `PI_ctrl_1` and `PI_ctrl_2`. At this stage, these identical subsystems, use identical data. If you have an Embedded Coder license, you can use these subsystems in an example that shows how to create reusable functions.

- 2 Open the `PI_ctrl_1` subsystem.



The PI controllers in the model are from a *library*, a group of related blocks or models for reuse. Libraries provide one of two methods for including and reusing models. The second method, model referencing, is described in “Simulation Test Environment” on page 19-6. You cannot edit a block that you add to a model from a library. Edit the block in the library so that instances of the block in different models remain consistent.

- 3 Open the Pos_Command_Arbitration subsystem. This Stateflow chart performs basic error checking on the two command signals. If the command signals are too far apart, the Stateflow diagram sets the output to a `fail_safe` position.



- 4 Close `throttlectrl`.

Simulation Test Environment

To test the throttle controller algorithm, you incorporate it into a *test harness*. A test harness is a model that evaluates the control algorithm and offers the following benefits:

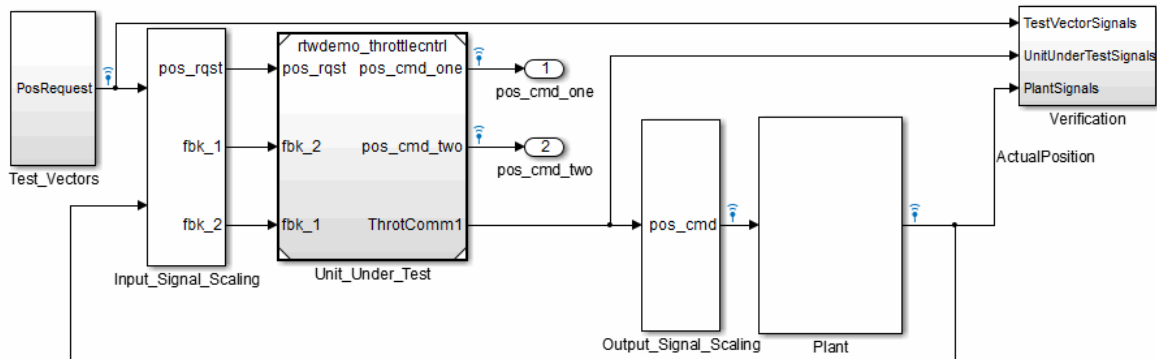
- Separates test data from the control algorithm.
- Separates the plant or feedback model from the control algorithm.
- Provides a reusable environment for multiple versions of the control algorithm.

The test harness model for this example implements a common simulation testing environment consisting of the following parts:

- Unit under test
- Test vector source
- Evaluation and logging
- Plant or feedback system
- Input and output scaling

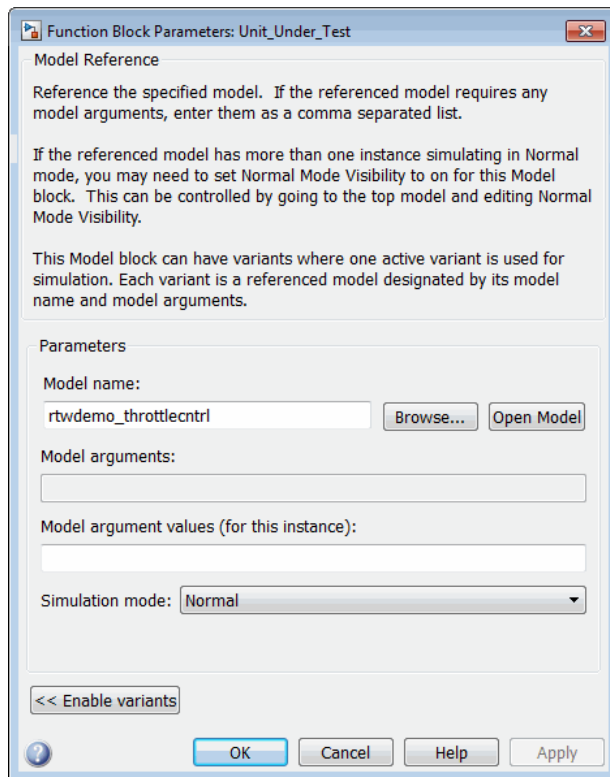
Explore the simulation testing environment.

- 1 Open the test harness model `rtwdemo_throttlecntrl_testharness` and save a copy as `throttlecntrl_testharness` in a writable location on your MATLAB path.



Copyright 2007-2011 The MathWorks, Inc.

- 2 Set up your `throttlecntrl` model as the control algorithm of the test harness.
 - a Open the `Unit_Under_Test` block and view the control algorithm.
 - b View the model reference parameters by right-clicking the `Unit_Under_Test` block and selecting **Block Parameters (ModelReference)**.



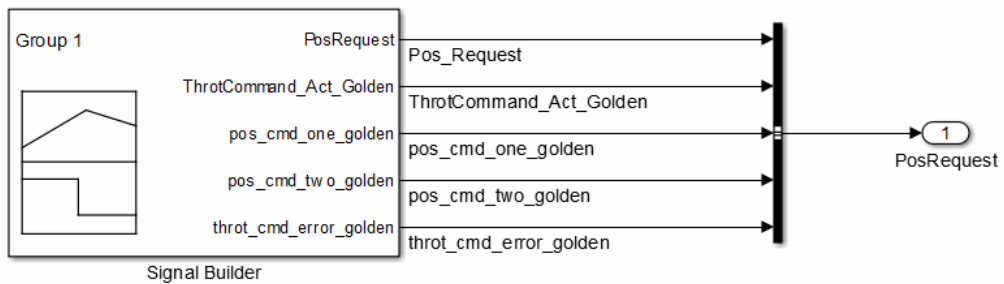
rtwdemo_throttlecntrl appears as the name of the referenced model.

- c Change the value of **Model name** to throttlecntrl.
- d Update the test harness model diagram by clicking **Simulation > Update Diagram**.

The control algorithm is the *unit under test*, as indicated by the name of the Model block, Unit_Under_Test.

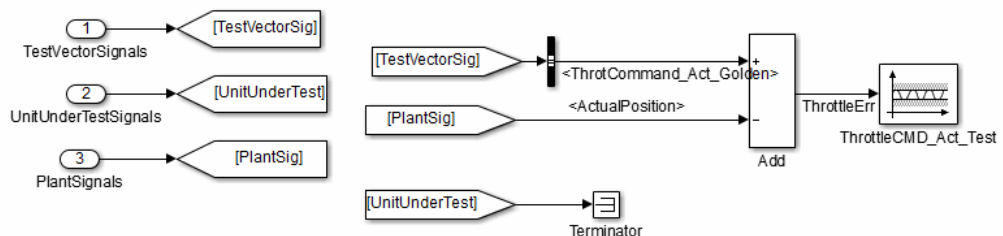
The Model block provides a method for reusing components. From the top model, it allows you to reference other models (directly or indirectly) as *compiled functions*. By default, Simulink software recompiles the model when the referenced models change. Compiled functions have the following advantages over libraries:

- Simulation time is faster for large models.
 - You can directly simulate compiled functions.
 - Simulation requires less memory. Only one copy of the compiled model is in memory, even when the model is referenced multiple times.
- 3 Open the *test vector source*, implemented in this test harness as the **Test_Vectors** subsystem.



The subsystem uses a Signal Builder block for the test vector source. The block has data that drives the simulation (**PosRequest**) and provides the expected results used by the **Verification** subsystem. This example test harness uses only one set of test data. Typically, you create a test suite that fully exercises the system.

- 4 Open the *evaluation and logging* subsystem, implemented in this test harness as subsystem **Verification**.

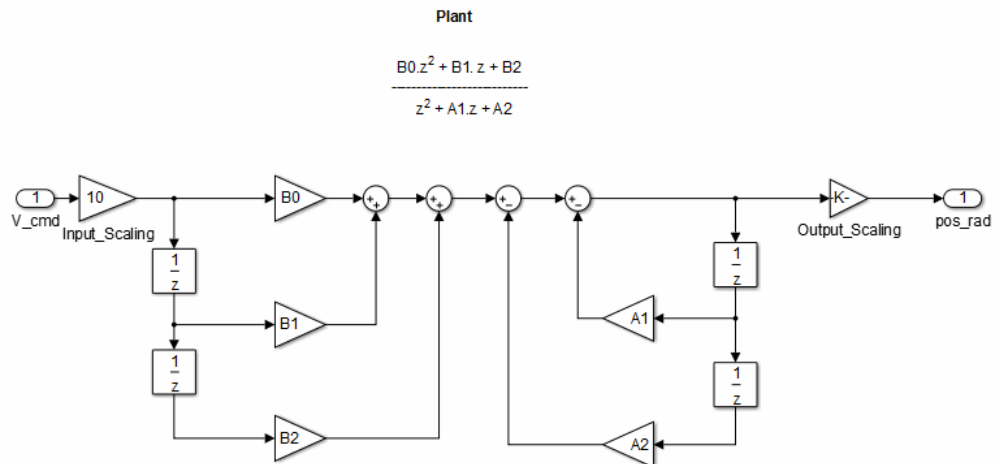


A test harness compares control algorithm simulation results against *golden data* — test results that exhibit the desired behavior for the control algorithm as certified by an expert. In the **Verification** subsystem, an Assertion block compares the simulated throttle value position from the plant against the golden value from the

test harness. If the difference between the two signals is greater than 5%, the test fails and the Assertion block stops the simulation.

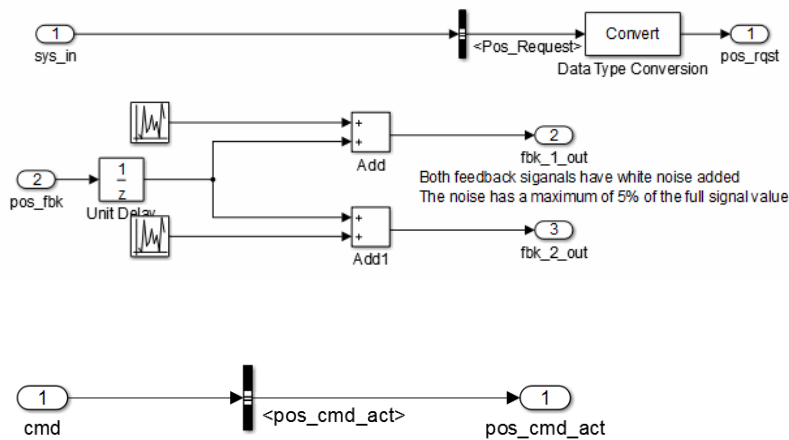
Alternatively, you can evaluate the simulation data after the simulation completes execution. Perform the evaluation with either MATLAB scripts or third-party tools. Post-execution evaluation provides greater flexibility in the analysis of data. However, it requires waiting until execution is complete. Combining the two methods can provide a highly flexible and efficient test environment.

- 5 Open the *plant or feedback system*, implemented in this test harness as the **Plant** subsystem.



The **Plant** subsystem models the throttle dynamics with a transfer function in canonical form. You can create plant models to varying levels of fidelity. It is common to use different plant models at different stages of testing.

- 6 Open the *input and output scaling* subsystems, implemented in this test harness as **Input_Signal_Scaling** and **Output_Signal_Scaling**.



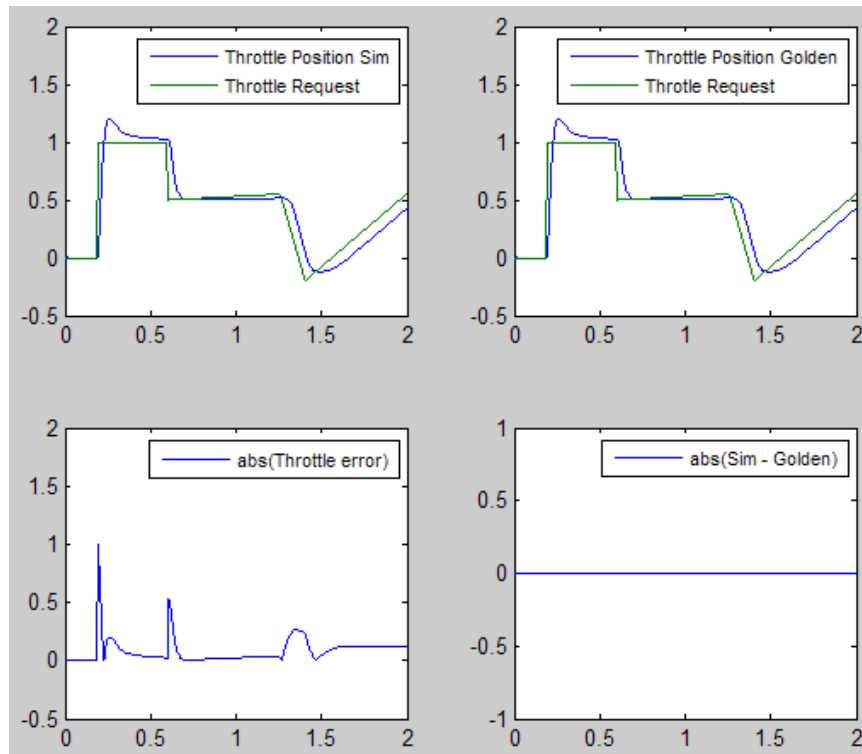
The subsystems that scale input and output perform the following primary functions:

- Select input signals to route to the unit under test.
- Select output signals to route to the plant.
- Rescale signals between engineering units and units that are writable for the unit under test.
- Handle rate transitions between the plant and the unit under test.

7 Save and close `throttlecntrl_testharness`.

Run Simulation Tests

- 1 Check that your working folder is set to a writable folder, such as the folder into which you placed copies of the example model files.
- 2 Open your copy of the test harness model, `throttlecntrl_testharness`.
- 3 Start a test harness model simulation. When the simulation is complete, the following results appear.



The lower-right hand plot shows the difference between the expected (golden) throttle position and the throttle position that the plant calculates. If the difference between the two values is greater than ± 0.05 , the simulation stops.

- 4 Save and close throttle controller and test harness models.

Key Points

- A basic model architecture separates calculations from signal routing and partitions the model into subsystems
- Two options for model reuse include block libraries and model referencing.
- If you represent your control algorithm in a test harness as a Model block, specify the name of the control algorithm model in the Model Reference Parameters dialog box.

- A test harness is a model that evaluates a control algorithm. Typically, a harness consists of a unit under test, a test vector source, evaluation and logging, a plant or feedback system, and input and output scaling components.
- The unit under test is the control algorithm being tested.
- The test vector source provides the data that drives the simulation which generates results used for verification.
- During verification, the test harness compares control algorithm simulation results against golden data and logs the results.
- The plant or feedback component of a test harness models the environment that is being controlled.
- When developing a test harness,
 - Scale input and output components.
 - Select input signals to route to the unit under test.
 - Select output signals to route to the plant.
 - Rescale signals between engineering units and units that are writable for the unit under test.
 - Handle rate transitions between the plant and the unit under test.
- Before running simulation or completing verification, consider checking a model with the Model Advisor.

Learn More

- “Support Model Referencing” on page 37-83
- “Code Generation”
- “Signal Groups”

Configure Model and Generate Code

In this section...

“About This Example” on page 19-14

“Configure the Model for Code Generation” on page 19-15

“Save Your Model Configuration as a MATLAB Function” on page 19-16

“Check Model Conditions and Configuration Settings” on page 19-17

“Generate Code for the Model” on page 19-17

“Review the Generated Code” on page 19-17

“Generate an Executable” on page 19-19

“Key Points” on page 19-19

About This Example

Learning Objectives

- Configure a model for code generation.
- Apply model checking tools to discover conditions and configuration settings resulting in generation of inaccurate or inefficient code.
- Generate code from a model.
- Locate and identify generated code files.
- Review generated code.

Prerequisites

- Ability to open and modify Simulink models and subsystems.
- Ability to set model configuration parameters.
- Ability to use the Simulink Model Advisor.
- Ability to read C code.
- An installed, supported C compiler.

Required Files

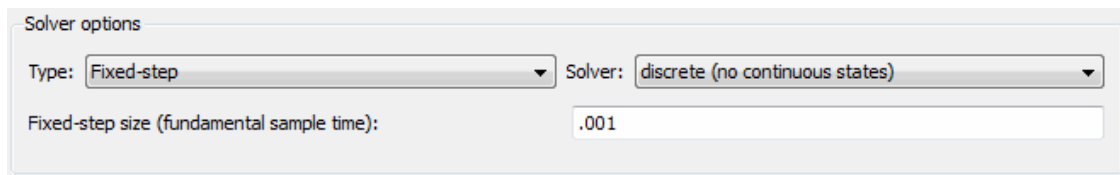
rtwdemo_throttlecntrl model file

Configure the Model for Code Generation

Model configuration parameters determine the method for generating the code and the resulting format.

- 1 Open `rtwdemo_throttlecntrl` and save a copy as `throttlecntrl` in a writable location on your MATLAB path.
- 2 Open the Configuration Parameters dialog box, **Solver** pane. To generate code for a model, you must configure the model to use a fixed-step solver. The following table shows the solver configuration for this example.

Parameter	Setting	Effect on Generated Code
Type	Fixed-step	Maintains a constant (fixed) step size, which is required for code generation
Solver	discrete (no continuous states)	Applies a fixed-step integration technique for computing the state derivative of the model
Fixed-step size	.001	Sets the base rate; must be the lowest common multiple of all rates in the system



- 3 Open the **Code Generation > General** pane and note that the **System target file** is set to `grt.tlc`.

Note: The GRT (Generic Real-Time Target) configuration requires a fixed-step solver. However, the `rsim.tlc` system target file supports variable step code generation.

The system target file (STF) defines a target, which is an environment for generating and building code for execution on a certain hardware or operating system platform. For example, one property of a target is the value for the `CodeFormat` TLC variable. The grt configuration requires a fixed step solver and the `rsim.tlc` supports variable step code generation.

- 4 Open the **Code Generation > Custom Code** pane and under **Include list of additional**, select **Include directories**. Note that the following path appears in the text field:

```
"$matlabroot$\toolbox\rtw\rtwdemos\EmbeddedCoderOverview\"
```

This folder includes files that are required to build an executable for the model.

- 5 Close the dialog box.

Save Your Model Configuration as a MATLAB Function

You can save the settings of model configuration parameters as a MATLAB function by using the `getActiveConfigSet` function. In the MATLAB Command Window, enter:

```
thcntrlAcs = getActiveConfigSet('throttlecntrl');  
thcntrlAcs.saveAs('throttlecntrlModelConfig');
```

You can then use the resulting function (for example, `throttlecntrlModelConfig`) to:

- Archive the model configuration.
- Compare different model configurations by using differencing tools.
- Set the configuration of other models.

For example, you can set the configuration of model `myModel` to match the configuration of the throttle controller model by opening `myModel` and entering:

```
myModelAcs = throttlecntrlModelConfig;  
attachConfigSet('myModel', myModelAcs, true);  
setActiveConfigSet('myModel', myModelAcs.Name);
```

For more information, see “Save a Configuration Set” and “Load a Saved Configuration Set” in the Simulink documentation.

Check Model Conditions and Configuration Settings

Before generating code for a model, use the Simulink Model Advisor to check the model for conditions and configuration settings. This check finds issues that can result in inaccurate or inefficient code.

- 1 Open `throttlecntrl`.
- 2 Start the Model Advisor by selecting **Analysis > Model Advisor > Model Advisor**. A dialog box opens showing the model system hierarchy.
- 3 Click `throttlecntrl` and then click **OK**. The Model Advisor window opens.
- 4 Expand **By Product** and **Embedded Coder**. By default, checks that do not trigger an Update Diagram, with one exception, are selected.
- 5 In the left pane, select the remaining checks and select **Embedded Coder**.
- 6 In the right pane, select **Show report after run** and click **Run Selected Checks**. The report shows a **Run Summary** that flags check warnings.
- 7 Review the report. The warnings highlight issues for embedded systems. At this point, you can ignore them. For more information about reports, see “View Model Advisor Reports” in the Simulink documentation.

Generate Code for the Model

- 1 Open `throttlecntrl`.
- 2 In the Configuration Parameters dialog box, select **Code Generation > Generate code only** and click **Apply**.
- 3 On the **Code Generation > Report** pane, select **Create code generation report** and click **Apply**.
- 4 In the model window, press **Ctrl+B** and watch the messages that appear in the MATLAB Command Window. The code generator produces standard C and header files, and an HTML code generation report. The code generator places the files in a *build folder*, a subfolder named `throttlecntrl_grt_rtw` under your current working folder.

Review the Generated Code

- 1 Open Model Explorer, and in the **Model Hierarchy** pane, expand the node for the `throttlecntrl` model, and select the **Code for** node.

- 2 In the **Contents** pane, select **HTML Report**. Model Explorer displays the HTML code generation report for the throttle controller model.
- 3 In the HTML report, click the link for the generated C model file and review the generated code. Note the following:
 - Identification, version, timestamp, and configuration comments.
 - Links to help you navigate within and between files
 - Data definitions
 - Scheduler code
 - Controller code
 - Model initialization and termination functions
 - Call interface for the GRT target — output, update, initialization, start, and terminate
- 4 Save and close `throttlecntrl`.

Consider examining the following files. In the HTML report **Contents** pane, click the links. Or, in your working folder, explore the generated code subfolder.

File	Description
<code>throttlecntrl.c</code>	C file that contains the scheduler, controller, initialization, and interface code
<code>throttlecntrl_data.c</code>	C file that assigns values to generated data structures
<code>throttlecntrl.h</code>	Header file that defines data structures
<code>throttlecntrl_private.h</code>	Header file that defines data used only by the generated code
<code>throttlecntrl_types.h</code>	Header file that defines the model data structure

For more information, see “Generated Source Files and File Dependencies” on page 19-38.

At this point, consider logging data to a MAT-file. For an example, see “Log Data for Analysis” on page 27-2.

Generate an Executable

- 1 Open `throttlectrl`.
- 2 In the Configuration Parameters dialog box, clear the **Code Generation > Generate code only** check box and click **Apply**.
- 3 Press **Ctrl+B**. Watch the messages in the MATLAB Command Window. The code generator uses a template make file associated with your target selection to create an executable that you can run on your workstation, independent of external timing and events.
- 4 Check your working folder for the `filethrottlectrl.exe`.
- 5 Run the executable. In the Command Window, enter `!throttlectrl`. The `!` character passes the command that follows it to the operating system, which runs the standalone program.

The program produces one line of output in the Command Window:

```
** starting the model **
```

At this point, consider logging data to a MAT-file. For an example, see “Log Data for Analysis” on page 27-2.

Tip For UNIX platforms, run the executable in the Command Window with the syntax `!./executable_name`. If preferred, run the executable from an OS shell with the syntax `./executable_name`. For more information, see “Run External Commands, Scripts, and Programs”.

Key Points

- To generate code change, the model configuration to specify a fixed-step solver and then select a system target format. Using the `grt.tlc` file requires a fixed-step solver. If the model contains continuous time blocks, you can use a variable-step solver with the `rsim.tlc` target.
- After debugging a model, consider configuring a model with parameter inlining enabled.
- Use the `getActiveConfigSet` function to save a model configuration for future use or to apply it to another model.
- Before generating code, consider checking a model with the Model Advisor.

- The code generator places generated files in a subfolder (*model_grt_rtw*) of your working folder.

More About

- “Code Generation”
- “Configuration Reuse”
- “Run Model Checks”

Configure Data Interface

About This Example

Learning Objectives

- Configure the data interface for code generated for a model.
- Control the name, data type, and data storage class of signals and parameters in generated code.

Prerequisites

- Understanding ways to represent and use data and signals in models.
- Familiarity with representing data constructs as data objects.
- Ability to read C code.

Required File

rtwdemo_throttlecntrl_datainterface model file

Declare Data

Most programming languages require that you *declare* data before using it. The declaration specifies the following information:

Data Attribute	Description
Scope	The region of the program that has access to the data
Duration	The period during which the data is resident in memory
Data type	The amount of memory allocated for the data
Initialization	An initial value, a pointer to memory, or NULL (if you do not provide an initial value, most compilers assign a zero value or a null pointer)

The following data types are supported for code generation.

Supported Data Types

Name	Description
double	Double-precision floating point
single	Single-precision floating point
int8	Signed 8-bit integer
uint8	Unsigned 8-bit integer
int16	Signed 16-bit integer
uint16	Unsigned 16-bit integer
int32	Signed 32-bit integer
uint32	Unsigned 32-bit integer
Fixed-point data types	8-, 16-, 32-bit word lengths

A *storage class* is the scope and duration of a data item. For more information about storage classes, see

- “Override Default Parameter Behavior by Creating Global Variables in the Generated Code” on page 14-45
- “Signal Storage Class” on page 14-111
- “Storage Classes for Block States” on page 14-123

Use Data Objects

In Simulink models and Stateflow charts, the following methods are available for declaring data: *data objects* and *direct specification*. This example uses the data object method. Both methods allow full control over the data type and storage class. You can mix the two methods in a single model.

In the MATLAB and Simulink environment, you can use data objects in various ways. This example focuses on the following types of data objects:

- Signal
- Parameter
- Bus

To configure the data interface for your model using the data object method, in the MATLAB base workspace, you define data objects. Then, associate them with your

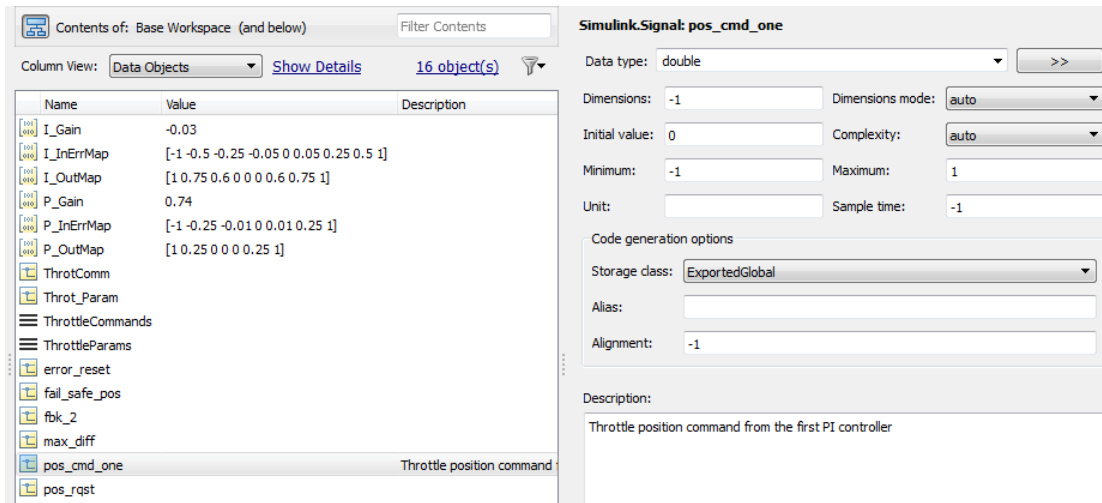
Simulink model or embedded Stateflow chart. When you build your model, the build process uses the associated base workspace data objects in the generated code.

You can set the values of the data object properties, which include:

- Data type
- Storage class
- Value (parameters)
- Initial value (signals)
- Alias (define a different name in the generated code)
- Dimension (typically inherited for parameters)
- Complexity (inherited for parameters)
- Unit (physical measurement unit)
- Minimum value
- Maximum value
- Description (used to document your data objects — does not affect simulation or code generation)

You can create and inspect base workspace data objects by entering commands in the MATLAB Command Window or by using Model Explorer. Perform the following steps to explore base workspace signal data objects.

- 1** Open `rtwdemo_throttlecntrl_datainterface` and save a copy as `throttlecntrl_datainterface` in a writable location on your MATLAB path.
- 2** Open Model Explorer.
- 3** Select **Base Workspace**.
- 4** Select the `pos_cmd_one` signal object for viewing.



You can also view the definition of a signal object. In the MATLAB Command Window, enter `pos_cmd_one`:

```
pos_cmd_one =
```

```
Signal with properties:
```

```

    CoderInfo: [1x1 Simulink.CoderInfo]
  Description: 'Throttle position command from the first PI controller'
    DataType: 'double'
         Min: -1
         Max: 1
         Unit: ''
    Dimensions: -1
  DimensionsMode: 'auto'
    Complexity: 'auto'
    SampleTime: -1
  InitialValue: '0'

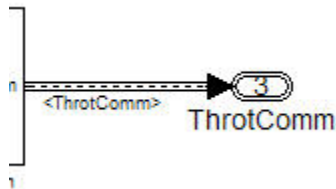
```

- 5 To view other signal objects, in Model Explorer, click the object name or in the MATLAB Command Window, enter the object name. The following table summarizes object characteristics for some of the data objects in this model.

Object Characteristics	pos_cmd_one	pos_rqst	P_InErrMap	ThrotComm*	ThrottleCommands*
Description	Top-level output	Top-level input	Calibration parameter	Top-level output structure	Bus definition
Data type	Double	Double	Auto	Auto	Structure
Storage class	Exported global	Imported extern pointer	Constant	Exported global	None

* `ThrottleCommands` defines a Bus object; `ThrotComm` is an instantiation of the bus. If the bus is a nonvirtual bus, the signal generates a structure in the C code.

You can use a bus definition (`ThrottleCommands`) to instantiate multiple instances of the structure. In a model diagram, a bus object appears as a wide line with central dashes, as shown.



Add New Data Objects

You can create data objects for named signals, states, and parameters. To associate a data object with a construct, the construct must have a name.

To find constructs for which you can create data objects, use the Data Object Wizard. This tool finds the constructs and then creates the objects for you. The model includes two signals that are not associated with data objects: `fbk_1` and `pos_cmd_two`.

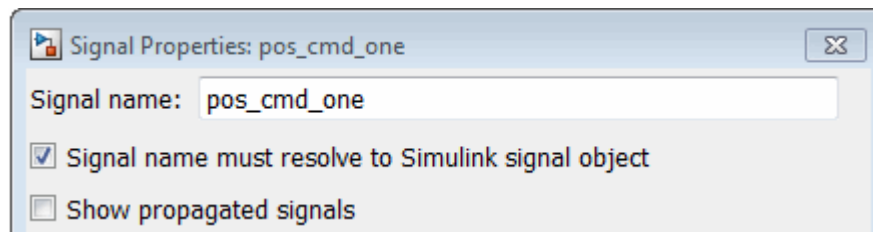
To find the signals and create data objects for them:

- 1 In the model window, select **Code > Data Objects > Data Object Wizard**. The Data Object Wizard dialog box opens.

- 2 To find candidate constructs, click **Find**. Constructs `fbk_1` and `pos_cmd_two` appear in the dialog box.
- 3 To select both constructs, click **Select All**.
- 4 In the table, under **Class**, make sure that each proposed data object uses the class `Simulink.Signal`. To change the class of the objects, click **Change Class**.
- 5 To create the data objects, click **Create**. Constructs `fbk_1` and `pos_cmd_two` are removed from the dialog box.
- 6 Close the Data Object Wizard.
- 7 In the **Contents** pane of the Model Explorer, find the newly created objects `fbk_1` and `pos_cmd_two`.

Enable Data Objects for Generated Code

- 1 Enable a signal to appear in generated code.
 - a In the model window, right-click the `pos_cmd_one` signal line and select **Properties**. A Signal Properties dialog box opens.
 - b Make sure that you select the **Signal name must resolve to Simulink signal object** parameter.



- 2 Enable signal object resolution for all signals in the model. In the MATLAB Command Window, enter:

```
disableimplicitsignalresolution('throttlectrl_datainterface')
```
- 3 Save and close `throttlectrl_datainterface`.

Effects of Simulation on Data Typing

In the throttle controller model, all data types are set to `double`. Because Simulink software uses the `double` data type for simulation, do not expect changes in the model

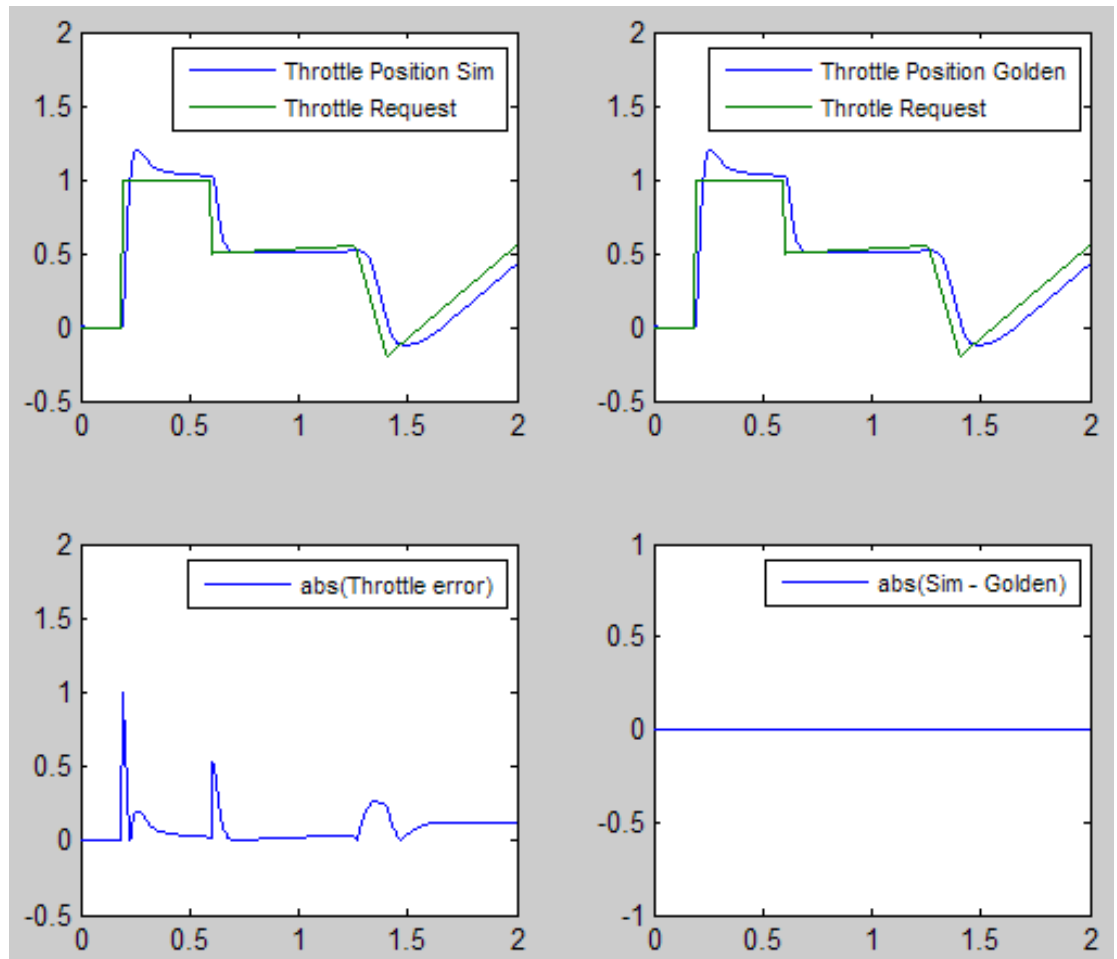
behavior when you run the generated code. You verify this effect by running the test harness.

Before you run your test harness, update it to include the `throttlecntrl_datainterface` model.

Note: The following procedure requires a Stateflow license.

- 1 Open `throttlecntrl_datainterface`.
- 2 Open your copy of test harness, `throttlecntrl_testharness`.
- 3 Right-click the `Unit_Under_Test Model` block and select **Block Parameters (ModelReference)**.
- 4 Set **Model name** to `throttlecntrl_datainterface`. Click **OK**.
- 5 Update the test harness model diagram.
- 6 Simulate the test harness.

The resulting plot shows that the difference between the golden and simulated versions of the model remains zero.



7 Save and close `throttlecntrl_testharness`.

Manage Data

Data objects exist in a separate file from the model in the base workspace. To save the data manually, in the MATLAB Command Window, enter `save`.

The separation of data from the model provides the following benefits:

- One model, multiple data sets:
 - Use of different parameter values to change the behavior of the control algorithm (for example, for reusable components with different calibration values)
 - Use of different data types to change targeted hardware (for example, for floating-point and fixed-point targets)
- Multiple models, one data set:
 - Sharing data between models in a system
 - Sharing data between projects (for example, transmission, engine, and wheel controllers can use the same CAN message data set)

Key Points

- You can declare data in Simulink models and Stateflow charts by using data objects or direct specification.
- From the Model Explorer or from the command line in the MATLAB Command Window, you manage (create, view, configure, and so on) base workspace data.
- The Data Object Wizard provides a quick way to create data objects for constructs such as signals, buses, and parameters.
- Configure data objects explicitly to appear by name in generated code.
- Separation of data from model provides several benefits.

More About

- “Load Signal Data for Simulation”
- “Data Representation”
- “Custom Storage Classes”
- “Manage Placement of Data Definitions and Declarations”

Call External C Functions

In this section...

“About This Example” on page 19-30

“Include External C Functions in a Model” on page 19-31

“Create a Block That Calls a C Function” on page 19-31

“Validate External Code in the Simulink Environment” on page 19-33

“Validate C Code as Part of a Model” on page 19-34

“Call a C Function from Generated Code” on page 19-36

“Key Points” on page 19-36

About This Example

Learning Objectives

- Evaluate a C function as part of a model simulation.
- Call an external C function from generated code.

Prerequisites

- Ability to open and modify Simulink models and subsystems.
- Ability to set model configuration parameters.
- Ability to read C code.
- An installed, supported C compiler.

Required Files

- `rtwdemo_throttlecntrl_extfunccall` model file
- `rtwdemo_ValidateLegacyCodeVrsSim` model file
- `/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files/SimpleTable.c`
- `/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files/SimpleTable.h`

Include External C Functions in a Model

Simulink models are one part of Model-Based Design. For many applications, a design also includes a set of pre-existing C functions created, tested (verified), and validated outside of a MATLAB and Simulink environment. You can integrate these functions easily into a model and the generated code. External C code can be used in the generated code to access hardware devices and external data files during rapid simulation runs.

This example shows you how to create a custom block that calls an external C function. Once the block is part of the model, you can take advantage of the simulation environment to test the system further.

Create a Block That Calls a C Function

To specify a call to an external C function, use an S-Function block. You can automate the process of creating the S-Function block by using the Simulink Legacy Code Tool. Using this tool, you specify an interface for your external C function. The tool then uses that interface to automate creation of an S-Function block.

- 1 Make copies of the files `SimpleTable.c` and `SimpleTable.h`, located in the folder `matlabroot/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files` (open). Put the copies in your working folder.
- 2 Create an S-Function block that calls the specified function at each time step during simulation:
 - a In the MATLAB Command Window, create a function interface definition structure:

```
def=legacy_code('initialize')
```

The data structure `def` defines the function interface to the external C code.

```
def =

    SFunctionName: ''
 InitializeConditionsFcnSpec: ''
    OutputFcnSpec: ''
    StartFcnSpec: ''
 TerminateFcnSpec: ''
    HeaderFiles: {}
    SourceFiles: {}
    HostLibFiles: {}
```

```
TargetLibFiles: {}
  IncPaths: {}
  SrcPaths: {}
  LibPaths: {}
SampleTime: 'inherited'
Options: [1x1 struct]
```

- b** Populate the function interface definition structure by entering the following commands:

```
def.OutputFcnSpec=['double y1 = SimpleTable(double u1,',...
  'double p1[], double p2[], int16 p3)'];
def.HeaderFiles = {'SimpleTable.h'};
def.SourceFiles = {'SimpleTable.c'};
def.SFunctionName = 'SimpTableWrap';
```

- c** Create the S-function:

```
legacy_code('sfcn_cmex_generate', def)
```

- d** Compile the S-function:

```
legacy_code('compile', def)
```

- e** Create the S-Function block:

```
legacy_code('slblock_generate', def)
```

A new model window opens that contains the `SimpTableWrap` block.

Tip Creating the S-Function block is a one-time task. Once the block exists, you can reuse it in multiple models.

- 3** Save the model to your working folder as: `s_func_simpltablewrap`.
4 Create a Target Language Compiler (TLC) file for the S-Function block:

```
legacy_code('sfcn_tlc_generate', def)
```

The TLC file is the component of an S-function that specifies how the code generator produces the code for a block.

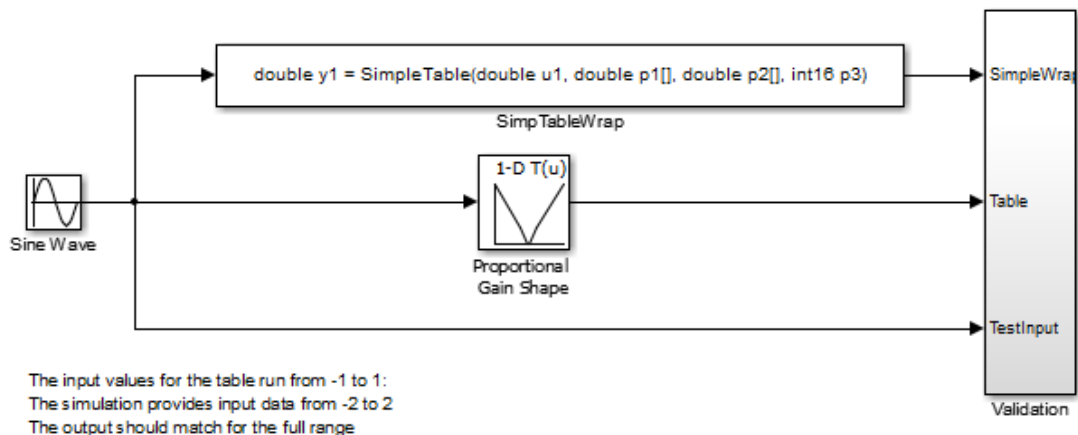
For more information on using the Legacy Code Tool, see:

- “Integrate C Functions Using Legacy Code Tool” in the Simulink documentation
- “Build S-Function With Legacy Code Tool” on page 24-27

Validate External Code in the Simulink Environment

When you integrate external C code with a Simulink model, before using the code, validate the functionality of the external C function code as a standalone component.

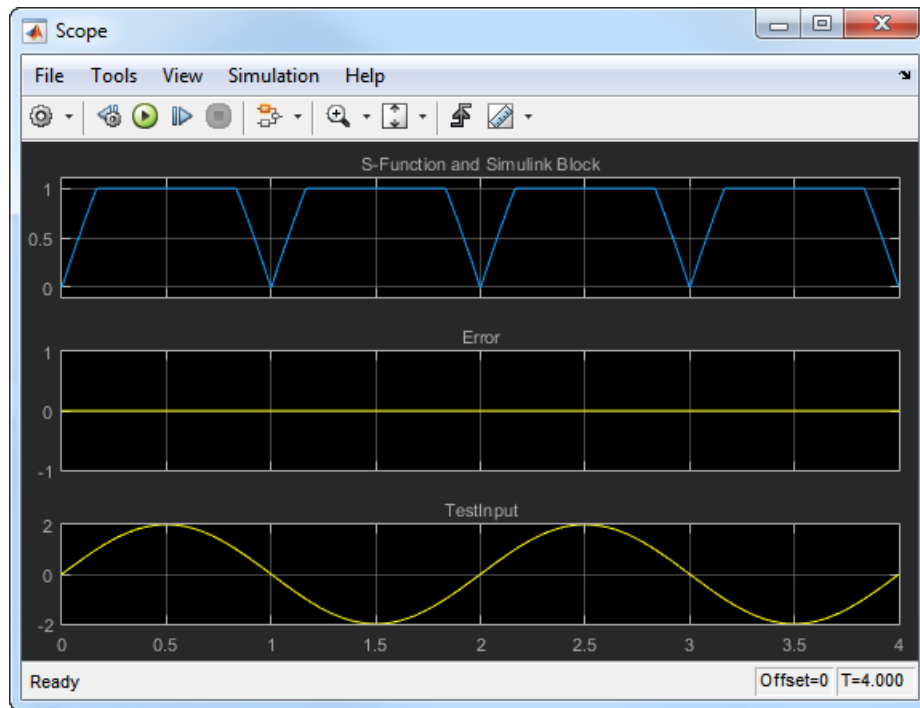
- 1 Open the model `rtwdemo_ValidateLegacyCodeVrsSim`. This model validates the S-function block that you created.



Copyright 2007-2011 The MathWorks, Inc.

- The **Sine Wave** block produces output values from $[-2 : 2]$.
 - The input range of the lookup table is from $[-1 : 1]$.
 - The output from the lookup table is the absolute value of the input.
 - The lookup table output clips the output at the input limits.
- 2 Simulate the model.
 - 3 View the validation results by opening the **Validation** subsystem and, in that subsystem, clicking the **Scope** block.

The following figure shows the validation results. The external C code and the Simulink Lookup table block provide the same output values.



- 4 Close the validation model.

Validate C Code as Part of a Model

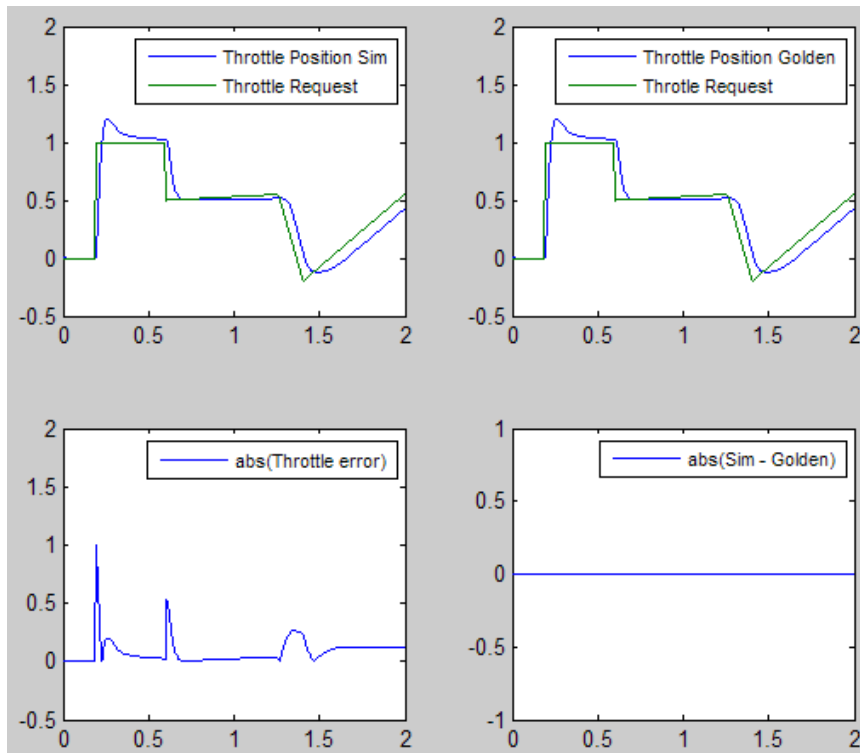
After you validate the functionality of the external C function code as a standalone component, validate the S-function in the model. Use the test harness model to complete the validation.

Note: The following procedure requires a Stateflow license.

- 1 Open `rtwdemo_throttlectrl_extfunccall` and save a copy to `throttlectrl_extfunccall` in a writable folder on your MATLAB path.
- 2 Examine the `PI_ctrl_1` and `PI_ctrl_2` subsystems.

- a Lookup blocks have been replaced with the block you created using the Legacy Code Tool.
 - b Note the block parameter settings for `SimpTableWrap` and `SimpTableWrap1`.
 - c Close the Block Parameter dialog boxes and the PI subsystem windows.
- 3 Open the test harness model, right-click the `Unit_Under_Test` Model block, and select **Block Parameters (ModelReference)**.
 - 4 Set **Model name** to `throttlecntrl_extfunccall`. Click **OK**.
 - 5 Update the test harness model diagram.
 - 6 Simulate the test harness.

The simulation results match the expected golden values.



- 7 Save and close `throttlecntrl_extfunccall` and `throttlecntrl_testharness`.

Call a C Function from Generated Code

The code generator uses a TLC file to process the S-Function block. Calls to C code embedded in an S-Function block:

- Can use data objects.
- Are subject to *expression folding*, an operation that combines multiple computations into a single output calculation.

- 1 Open `throttlecntrl_extfunccall`.
- 2 Generate code for the model.
- 3 Examine the generated code in the file `throttlecntrl_extfunccall.c`.
- 4 Close `throttlecntrl_extfunccall` and `throttlecntrl_testharness`.

Key Points

- You can easily integrate external functions into a model and generated code by using the Legacy Code Tool.
- Validate the functionality of external C function code which you integrate into a model as a standalone component.
- After you validate the functionality of external C function code as a standalone component, validate the S-function in the model.

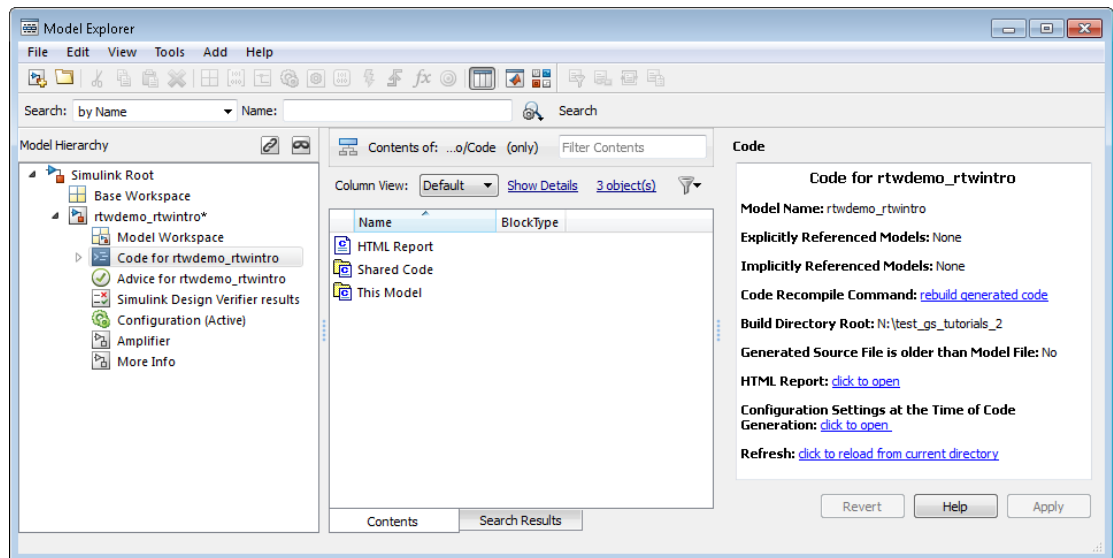
More About

- “Integrate C Functions Using Legacy Code Tool”
- “S-Functions and Code Generation” on page 24-99

Reload Generated Code

You can reload the code generated for a model from the Model Explorer.

- 1 Click the **Code for *model*** node in the **Model Hierarchy** pane.
- 2 In the **Code** pane, click the **Refresh** link.



The code generator reloads the code for the model from the build folder.

More About

- “Rebuild a Model” on page 25-41
- “Control Regeneration of Top Model Code” on page 25-43

Generated Source Files and File Dependencies

The code generator produces code in a set of source files that vary little among different system target files. Not all possible files are generated for every model. Some files are created only when the model includes subsystems, calls external interfaces, or uses particular types of data. The code generator handles most of the code formatting decisions (such as identifier construction and code packaging) in consistent ways.

In this section...
“About Generated Files and File Dependencies” on page 19-38
“Dependencies When Interfacing Custom and Generated Code” on page 19-39
“Dependencies of the Generated Code” on page 19-49
“Specify Include Paths in Generated Source Files” on page 19-54

About Generated Files and File Dependencies

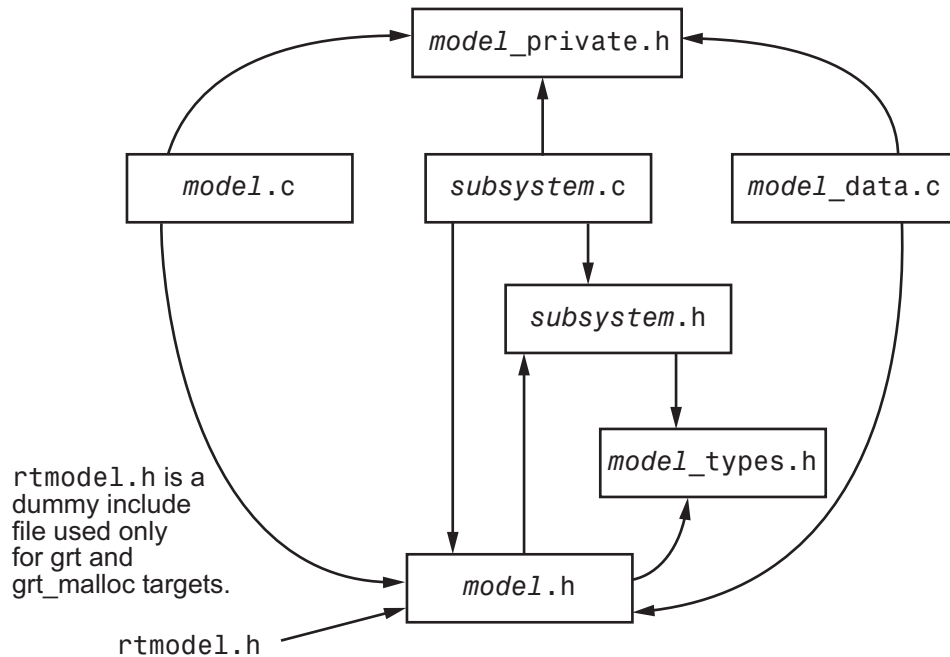
The source and make files created during the build process are generated into your build folder, which is created or reused in your current folder. Some files are unconditionally generated, while the existence of others depends on target settings and options (for example, support files for C API or external mode). See “Files and Folders Created During Build Process” on page 19-57 for descriptions of the generated files.

Note The file packaging of ERT based system target files differs slightly from the file packaging as described. See “Generate Code Modules” in the Embedded Coder documentation for more information.

Generated source file dependencies are depicted in the next figure. Arrows coming from a file point to files it includes. Other dependencies exist, for example on Simulink header files `tmwtypes.h` and `simstruc_types.h`, plus C or C++ library files. The figure maps inclusion relations between only those files that are generated in the build folder. These files can also reference utility and model reference code located in a code generation folder. See “Code Generation Folder Structure for Model Reference Targets” on page 8-15 for details.

The figure shows that parent system header files (`model.h`) include child subsystem header files (`subsystem.h`). In more layered models, subsystems similarly include their

children's header files, on down the model hierarchy. As a consequence, subsystems are able to recursively “see” into their descendant subsystems and see into the root system (because every *subsystem.c* or *subsystem.cpp* includes *model.h* and *model_private.h*).



Generated File Dependencies

Note: In the preceding figure, files *model.h*, *model_private.h*, and *subsystem.h* also depend on the header file *rtwtypes.h*. Targets that are not based on the ERT system target file can have additional dependencies on *tmwtypes.h* and *simstruc_types.h*.

Dependencies When Interfacing Custom and Generated Code

You can incorporate legacy or custom code to build with generated code in several ways. One common approach is by creating S-functions. For details on this approach, see “S-Functions and Code Generation” on page 24-99.

Another approach is to interface code using global variables created by declaring storage classes for signals and parameters. This approach requires customizing an outer code harness, typically referred to as a `main.c` or `main.cpp` file, properly to execute with the generated code. In addition, the harness can contain custom code.

These scenarios require including header files that are specific to the code generator product. These inclusions make available function declarations, type definitions, and defines to the legacy or custom code.

rtwtypes.h

The header file `rtwtypes.h` defines data types, structures, and macros required by the generated code. Normally, you should include `rtwtypes.h` for both GRT and ERT targets instead of including `tmwtypes.h` or `simstruc_types.h`. However, the contents of the header file varies depending on your target selection.

For...	<code>rtwtypes.h</code>
GRT target	Provides a complete set of definitions by including <code>tmwtypes.h</code> and <code>simstruc_types.h</code> , both of which depend on <ul style="list-style-type: none"> • System header files <code>limits.h</code> and <code>float.h</code> • Code generator header files: <code>rtw_matlogging.h</code>, <code>rtw_extmode.h</code>, <code>rtw_continuous.h</code>, and <code>rtw_solver.h</code>
ERT target and targets based on the ERT target	Is optimized, when possible, to include a minimum set of <code>#define</code> statements, enumerations, and type definitions; does not include <code>tmwtypes.h</code> and <code>simstruc_types.h</code>

The code generator produces the optimized version of `rtwtypes.h` for ERT based system target files when both of the following conditions exist:

- The **Classic call interface** option on the **All Parameters** tab of the Configuration Parameters dialog box is cleared.
- The model does not contain noninlined S-functions

Include `rtwtypes.h`. If you include it for GRT targets, for example, it is easier to use your code with ERT-based targets.

For GRT and ERT targets, the location of `rtwtypes.h` depends on whether the build uses the *shared utilities* location. If you use a shared location, the code generator places `rtwtypes.h` in `slprj/target/_sharedutils`; otherwise, it places `rtwtypes.h` in

the standard build folder (*model_target_rtw*). See “Specify Target Interfaces” on page 17-10 for more information on when and how to use the shared utilities location.

The header file `rtwtypes.h` should be included by source files that use code generator type names or other code generator definitions. A typical example is for files that declare variables using a code generator data type, for example, `uint32_T myvar;`.

A source file that is intended to be used by the code generator and by an S-function can use the preprocessor macro `MATLAB_MEX_FILE`. The macro definition comes from the `mex` function:

```
#ifndef MATLAB_MEX_FILE
#include "tmwtypes.h"
#else
#include "rtwtypes.h"
#endif
```

A source file meant to be used as the code generator `main.c` (or `.cpp`) file also includes `rtwtypes.h` without preprocessor checks.

```
#include "rtwtypes.h"
```

Custom source files that are generated using the Target Language Compiler can also emit these `include` statements into their generated file.

The includes for `rtwtypes.h` and other generated header files can appear in many source files. To prevent multiple inclusions, these header files have preprocessor definitions and checks. To avoid name collision, the preprocessor definitions have double leading and trailing underscores. An example preprocessor check using one of these definitions is:

```
#ifndef __RTWTYPES_H__
#define __RTWTYPES_H__
```

For more information on the `rtwtypes.h` file, see “`rtwtypes.h` and Shared Utility Code” on page 19-81.

multiword_types.h

The header file `multiword_types.h` contains type definitions for wide data types and their chunks. If your code uses multiword data types, include this header file.

model.h

The header file `model.h` declares model data structures and a public interface to the model entry points and data structures. This header file also provides an interface to the

real-time model data structure (*model_M*) by using access macros. If your code interfaces to model functions or model data structures, include *model.h*:

- Exported global signals

```
extern int32_T INPUT;    /* '<Root>/In' */
```

- Global structure definitions

```
/* Block parameters (auto storage) */  
extern Parameters_mymodel mymodel_P;
```

- RTM macro definitions

```
#ifndef rtmGetSampleTime  
# define rtmGetSampleTime(rtm, idx)  
((rtm)->Timing.sampleTimes[idx])  
#endif
```

- Model entry point functions (ERT example)

```
extern void mymodel_initialize(void);  
extern void mymodel_step(void);  
extern void mymodel_terminate(void);
```

The *main.c* (or *.cpp*) file of a code generation target should include *model.h*. If the *main.c* (or *.cpp*) file is generated from a TLC script, the TLC source can include *model.h* using:

```
#include "%<CompiledModel.Name>.h"
```

If *main.c* is a static source file, the fixed header file name *rtmodel.h* can be used. This file includes the *model.h* header file:

```
#include "model.h"    /* If main.c is generated */
```

or

```
#include "rtmodel.h" /* If static main.c is used */
```

Other custom source files might need to include *model.h* to interface to model data, for example exported global parameters or signals. The *model.h* file itself can have additional header dependencies, as listed in the tables System Header Files and Code Generator Header Files, due to requirements of generated code.

Tip To reduce dependencies (which can reduce the number of included header files), the information in the System Header Files table and Code Generator Header Files table provides a starting point. Some header files (for example, <stddef.h>) are included for broadly defined cases (for example, included when the model contains a utility function that needs it). You can use the following approach to help identify header file dependencies (such as utilities that depend on <stddef.h>):

- Set the Shared code placement parameter to 'Shared location' and build the model. The code generator places the utility functions in __sharedutils folder.
- Use a grep utility to search the .c and .h files in the __sharedutils folder for #include. The search results list the utilities with header file dependencies (such as <stddef.h>).
- Use this information to identify utilities to remove from the model and reduce header file dependencies in the generated code.

For more information, see “Shared Fixed-Point Utility Functions” on page 19-88.

System Header Files

Header File	Purpose	GRT Targets	ERT Targets
<float.h>	Defines math constants	Not included	Included when code generator honors the Stop time solver configuration parameter due to one of the following options being set: <ul style="list-style-type: none"> • All Parameters > MAT-file logging • Code Generation > Interface > External mode
<math.h>	Provides floating-point math functions	Included when the model contains a floating-point math function	Included when the model contains a floating-point math function that is not overridden by a code

Header File	Purpose	GRT Targets	ERT Targets
			replacement library entry For more information, see “Choose a Code Replacement Library” on page 21-8.
<stddef.h>	Defines NULL	Included when the model contains a utility function that needs it	Included when the model contains a utility function that needs it
<stdio.h>	Provides file I/O functions	Included when the model includes a TO File block	Included when the model includes a TO File block, or you select Configuration Parameters > All Parameters > MAT-file logging . See “MAT-file logging”.
<stdlib.h>	Provides utility functions such as <code>div()</code> and <code>abs()</code>	Included when the model includes <ul style="list-style-type: none"> • A Stateflow chart • A Math Function block configured for <code>mod()</code> or <code>rem()</code>, which generate calls to <code>div()</code> 	Included when the model includes <ul style="list-style-type: none"> • A Stateflow chart and you select the Support: floating-point numbers configuration parameter • A Math Function block configured for <code>mod()</code> or <code>rem()</code>, which generate calls to <code>div()</code>
<string.h>	Provides memory functions such as <code>memset()</code> and <code>memcpy()</code>	Included due to use of <code>memset()</code> in model initialization code	Included when block or model initialization code calls <code>memcpy()</code> or <code>memset()</code>

Header File	Purpose	GRT Targets	ERT Targets
			<p>For a list of relevant blocks, enter <code>showblockdatatypetable</code> in the MATLAB Command Window and look for blocks with the N2 note.</p> <p>To omit calls to <code>memset()</code> from model initialization code, select the Remove root level I/O zero initialization and Remove internal data zero initialization optimization configuration parameters.</p>

Header file dependencies in the Code Generator Header Files table apply to the system target files `grt.tlc` and `ert.tlc`. Targets derived from these base targets may have additional header dependencies. Also, code generation for blocks from blocksets, embedded targets, and custom S-functions may introduce additional header dependencies.

Code Generator Header Files

Header File	Purpose	GRT Targets	ERT Targets
<code>builtin_typeid_type</code>	Defines an enumerated type corresponding to built-in data types	Included when your model contains a Stateflow chart that uses messages or when you select one or more of the following in the Configuration Parameters dialog box:	Included when your model contains a Stateflow chart that uses messages or when you select one or more of the following in the Configuration Parameters dialog box:

Header File	Purpose	GRT Targets	ERT Targets
		<ul style="list-style-type: none"> • All Parameters > MAT-file logging • Any C API option at Code Generation > Interface 	<ul style="list-style-type: none"> • MAT-file logging • Any C API option
dt_info.h	Defines data structures for external mode	Included when you configure a model for external mode	Included when you configure a model for external mode
ext_work.h	Defines external mode functions	Included when you configure a model for external mode	Included when you configure a model for external mode
fixedpoint.h	Provides fixed-point support for noninlined S-functions	Included	<p>Included when either of the following conditions exists:</p> <ul style="list-style-type: none"> • The model uses noninlined S-functions • You select the configuration parameter Classic call interface
model_reference_typ	Contains type definitions for timing bridges	Included for a model reference target or a model containing model reference blocks	Included for a model reference target or a model containing model reference blocks
model_types.h	Defines model-specific data types	Included	Included
multiword_types.h	Contains type definitions for wide data types and their chunks	Included when multiword data types are used. Included when you select one or more of the following in the Configuration Parameters dialog box:	Included when multiword data types are used. Included when you select one or more of the following in the Configuration Parameters dialog box:

Header File	Purpose	GRT Targets	ERT Targets
		<ul style="list-style-type: none"> • All Parameters > MAT-file logging • Code Generation > Interface > External mode 	<ul style="list-style-type: none"> • All Parameters > MAT-file logging • Code Generation > Interface > External mode
rt_defines.h	Contains type definitions for special mathematical constants (such as π and e) and defines the <code>UNUSED_PARAMETER</code> macro	Included when a mathematical constant definition is required or when a model function argument is required but not accessed by the function body	Included when a mathematical constant definition is required or when a model function argument is required but not accessed by the function body
rt_logging.h	Supports MAT-file logging; also includes <code>rtwtypes.h</code> , <code>builtin_typeid_types.h</code> , <code>multiword_types.h</code> , <code>rt_mxclassid.h</code> , and <code>rtw_matlogging-.h</code>	Included	Included when you open the Configuration Parameters dialog box and select All Parameters > MAT-file logging . See “MAT-file logging”.
rt_mxclassid.h	Defines <code>mxArray</code> class ID enumerations	Included by <code>rt_logging.c</code>	Included by <code>rt_logging.c</code>
rt_nonfinite.h rtGetInf.h rtGetNaN.h	Support nonfinite numbers	<p>Conditions under which these files are included include the following:</p> <ul style="list-style-type: none"> • The generated code requires nonfinite numbers. • You select Configuration Parameters > All Parameters > MAT-file logging. • The model contains S-functions. 	<p>Conditions under which these files are included include the following:</p> <ul style="list-style-type: none"> • The generated code requires nonfinite numbers. • You select Configuration Parameters > All Parameters > MAT-file logging. • The model contains S-functions.

Header File	Purpose	GRT Targets	ERT Targets
		<ul style="list-style-type: none"> You select the model option Classic call interface for a GRT model. 	
rtw_continuous.h	Supports continuous time	Included by <code>simstruc_types.h</code>	Included when you select the configuration parameter Support: continuous time and <code>simstruc.h</code> is not already included
rtw_extmode.h	Supports external mode	Included by <code>simstruc_types.h</code>	Included when you configure the model for external mode and <code>simstruc.h</code> is not already included
rtw_matlogging.h	Supports MAT-file logging	Included by <code>simstruc_types.h</code> and <code>rt_logging.h</code>	Included by <code>rt_logging.h</code>
rtw_solver.h	Supports continuous states	Included by <code>simstruc_types.h</code>	Included when you select the configuration parameter Support: continuous time and <code>simstruc.h</code> is not already included
rtwtypes.h	Defines code generator data types; generated file	Included; use the complete version of the file, which includes <code>tmwtypes.h</code> and <code>simstruc_types.h</code> (see <code>simstruc_types.h</code> for dependencies)	Included; use the complete or optimized version of the file as explained in “rtwtypes.h” on page 19-40
simstruc.h	Provides support for calling noninlined S-functions that use the <code>Simstruct</code>	Included	Included when either of the following conditions exists:

Header File	Purpose	GRT Targets	ERT Targets
	definition; also includes <code>limits.h</code> , <code>string.h</code> , <code>tmwtypes.h</code> , and <code>simstruc_types.h</code>		<ul style="list-style-type: none"> The model uses noninlined S-functions. You select the configuration parameter Classic call interface.
<code>simstruc_types.h</code>	Provides definitions used by generated code and includes the header files <code>rtw_matlogging.h</code> , <code>rtw_extmode.h</code> , <code>rtw_continuous.h</code> , <code>rtw_solver.h</code> , and <code>sysran_types.h</code>	Included with <code>rtwtypes.h</code>	Not included; <code>rtwtypes.h</code> contains definitions and <code>model.h</code> contains header files
<code>sysran_types.h</code>	Supports external mode	Included by <code>simstruc_types.h</code>	Included when you configure the model for external mode and <code>simstruc.h</code> is not already included

Dependencies of the Generated Code

When you specify a GRT based system target file, the code generator can build standalone executable program than can run on the development computer. Several processor- and operating system-specific targets also provide automated builds using a cross-compiler. These targets are typically makefile-based interfaces for which the code generator provides a “Template MakeFile (TMF) to makefile” conversion capability. Part of this conversion process is to include, in the generated makefile, source file, header file, and library file information (the dependencies) for a compilation.

In other instances, the generated model code needs to be integrated into a specific application. Or, it may be desired to enter the generated files and file dependencies into a configuration management system. This section discusses the various aspects of the generated code dependencies and how to determine them.

Typically, the generated code for a model consists of a small set of files:

- `model.c` or `model.cpp`
- `model.h`
- `model_data.c` or `model_data.cpp`
- `model_private.h`
- `rtwtypes.h`

These files are generated in the build folder for a standalone model or a subfolder under the `slprj` folder for a model reference target. There is also a top-level `main.c` (or `.cpp`) file that calls the top model functions to execute the model. The main program is either a static file, such as the `rt_main.c` file provided by the software, or, for ERT-based targets, can be dynamically generated.

The preceding files also have dependencies on other files, which occur due to:

- Including other header files
- Using macros declared in other header files
- Calling functions declared in other source files
- Accessing variables declared in other source files

These dependencies are introduced for various reasons such as:

- Blocks in a model generate code that makes function calls. These calls can occur in several forms:
 - The called functions are declared in other source files. In some cases such as a blockset, these source file dependencies are typically managed by compiling them into a library file.
 - In other cases, the called functions are provided by the compilers own run-time library, such as for functions in the ANSI⁵ C header, `math.h`.
 - Some function dependencies are themselves generated files. Some examples are for fixed-point utilities and nonfinite support. These dependencies are referred to as shared utilities. The generated functions can appear in files in the build folder for standalone models or in the `_sharedutils` folder under the `slprj` folder for builds that involve model reference.
- Models with continuous time require solver source code files.

5. ANSI is a registered trademark of the American National Standards Institute, Inc.

- Code generator options such as external mode, C API, and MAT-file logging trigger additional dependencies.
- Specifying custom code can introduce dependencies.

Providing the Dependencies

The code generator provides several mechanisms for feeding file dependency information into the build process. The mechanisms available to you depend on whether your dependencies are block based or are model or target based.

For block dependencies, consider using

- S-functions and blocksets
 - folders that contain S-function MEX-files used by a model are added to the header include path.
 - Makefile rules are created for these folders to allow source code to be found.
 - For S-functions that are not inlined with a TLC file, the S-function source file name is added to the list of sources to compile.
 - The S-Function block parameter `SFunctionModules` lets you specify additional source file names.
 - The `rtwmakecfg.m` mechanism provides further capability in specifying dependencies. See “Use `rtwmakecfg.m` API to Customize Generated Makefiles” on page 35-24 for more information.

For more information on applying these approaches to legacy or custom code integration, see “Build S-Function With Legacy Code Tool” on page 24-27.

- S-Function Builder block, which provides its own GUI for specifying dependency information

For model- or target-based dependencies, such as custom header files, consider using

- The **Code Generation > Custom Code** pane of the Configuration Parameters dialog box, which lets you specify additional libraries, source files, and include folders.
- TLC functions `LibAddToCommonIncludes()` and `LibAddToModelSources()`, which allow you to specify dependencies during the TLC phase. See “`LibAddToCommonIncludes(incFileName)`” and “`LibAddSourceFileCustomSection(file, builtInSection, newSection)`” in the Target Language Compiler documentation for details. The Embedded Coder product also

provides a TLC-based customization template capability for generating additional source files.

Makefile Considerations

In most cases, system target files are makefile based. A “Template MakeFile (TMF) to makefile” conversion capability is available. The template makefile contains a token expansion mechanism in which the build process expands different tokens in the makefile to include the additional dependency information. The resulting makefile contains the complete dependency information. See “Customize Template Makefiles” on page 37-62 for more information on working with template makefiles.

The generated makefile contains the following information:

- Names of the source file dependencies (by using various **SRC** variables)
- folders where source files are located (by using unique rules)
- Location of the header files (by using the **INCLUDE** variables)
- Precompiled library dependencies (by using the **LIB** variables)
- Libraries which need to be compiled and created (by using rules and the **LIB** variables)

A property of make utilities is that the specific location for a given source C or C++ file does not need to be specified. If a rule exists for that folder and the source file name is a prerequisite in the makefile, the make utility can find the source file and compile it. Similarly, the C or C++ compiler (preprocessor) does not require absolute paths to the headers. Given the name of header file by using an `#include` directive and an include path, it is able to find the header. The generated C or C++ source code depends on this standard compiler capability.

Also, libraries are typically created and linked against, but occlude the specific functions that are being used.

Although the build process succeeds and can create a minimum-size executable, these properties can make it difficult to determine manually the minimum list of file dependencies along with their fully qualified paths. The makefile can be used as a starting point to determining the dependencies that the generated model code has.

An additional approach to determining the dependencies is by using linker information, such as a linker map file, to determine the symbol dependencies. The location of code generator and blockset source and header files is provided as shown to help in locating the dependencies.

Code Generator Static File Dependencies

Several locations in the MATLAB folder tree contain static file dependencies specific to the code generator:

- `matlabroot/rtw/c/src` (open)

This folder has subfolders and contains additional files that may need to be compiled. Examples include solver functions (for continuous time support), external mode support files, C API support files, and S-function support files. Source files in this folder are included into the build process using in the SRC variables of the makefile.

- Header files in the folder `fullfile(matlabroot, '/rtw/extern/include')`
- Header files in the folder `fullfile(matlabroot, '/simulink/include')`

These folders contain additional header file dependencies such as `tmwtypes.h`, `simstruc_types.h`, and `simstruc.h`.

Note For ERT-based targets, several header dependencies from the shown locations can be avoided. ERT-based targets generate the minimum set of type definitions, macros, and so on, in the file `rtwtypes.h`.

Blockset Static File Dependencies

Blockset products apply the `rtwmakecfg.m` mechanism to provide the code generator with dependency information. As such, the `rtwmakecfg.m` file provided by the blockset contains the listing of include path and source path dependencies for the blockset. Typically, blocksets create a library from the source files, which the generated model code can then link against. The libraries are created and identified using the `rtwmakecfg.m` mechanism.

To locate the `rtwmakecfg.m` files for blocksets in your MATLAB installed tree, use the following command:

```
>> which -all rtwmakecfg.m
```

For example, for the DSP System Toolbox product, the `which` command displays a path similar to the following:

```
C:\Program Files\MATLAB\toolbox\dspblks\dspmex\rtwmakecfg.m
```

If the model being compiled uses one or more of the blocksets listed by the `which` command, you can determine folder and file dependency information from the respective

rtwmakecfg.m file. For example, here is an excerpt of the rtwmakecfg.m file for the DSP System Toolbox product.

```
function makeInfo=rtwmakecfg()
%RTWMAKECFG adds include and source directories to RTW make files.
% makeInfo=RTWMAKECFG returns a structured array containing build info.
% Please refer to the rtwmakecfg API section in the Simulink Coder
% documentation for details on the format of this structure.
.
.
.
makeInfo.includePath = { ...
    fullfile(matlabroot,'toolbox','dspblks','include') };
makeInfo.sourcePath = {};
```

Specify Include Paths in Generated Source Files

You can add `#include` statements to generated code. Such references can come from several sources, including TLC scripts for inlining S-functions, custom storage classes, bus objects, and data type objects. The included files typically consist of header files for legacy code or other customizations. Also, you can specify compiler include paths with the `-I` compiler option. The build process uses the specified paths to search for included header files.

Usage scenarios for the generated code include, but are not limited to, the following:

- Generated code is compiled with a custom build process that requires an environment-specific set of `#include` statements.

In this scenario, the code generator is likely invoked with the **Generate code only** check box selected. Consider using fully qualified paths, relative paths, or just the header file names in the `#include` statements, and also use include paths.

- The generated code is compiled using the build process.

In this case, you can specify compiler include paths (`-I`) for the build process in several ways:

- The **Code Generation > Custom Code** pane of the Configuration Parameters dialog box allows you to specify additional include paths. The include paths are propagated into the generated makefile when the template makefile (TMF) is converted to the actual makefile.

- The `rtwmakecfg.m` mechanism allows S-functions to introduce additional include paths into the build process. The include paths are propagated when the template makefile (TMF) is converted to the actual makefile.
- When building a custom target that is makefile-based, the desired include paths can be directly added into the targets template makefile.
- Use the `make` command to specify a `USER_INCLUDES` make variable that defines a folder in which the build process should search for included files. For example,

```
make_rtw USER_INCLUDES=-Id:\work\feature1
```

The custom includes are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler.

Recommended Approaches

The following are recommended approaches for using `#include` statements and include paths with the build process to generate code that remains portable and minimizes compatibility problems with future versions.

Assume that additional header files are:

```
c:\work\feature1\foo.h
c:\work\feature2\bar.h
```

- A recommended approach is to include in the `#include` statements only the file name, such as:

```
#include "foo.h"
#include "bar.h"
```

Then, the include path passed to the compiler contains folders in which the headers files exist:

```
cc -Ic:\work\feature1 -Ic:\work\feature2 ...
```

- A second recommended approach is to use relative paths in `#include` statements and provide an anchor folder for these relative paths using an include path, for example,

```
#include "feature1\foo.h"
#include "feature2\bar.h"
```

Then specify the anchor folder (for example `\work`) to the compiler:

```
cc -Ic:\work ...
```

Folder Dependencies to Avoid

When using the build process, avoid dependencies on folders in its code generation folder, such as the *model_ert_rtw* folder or the *slprj* folder. Thus, the `#include` statements should not just be relative to where the generated source file exists. For example, if your MATLAB code generation folder is `c:\work`, a generated *model.c* source file would be generated into a subfolder such as:

```
c:\work\model_ert_rtw\model.c
```

The *model.c* file would have `#include` statements of the form

```
#include "..\feature1\foo.h"  
#include "..\feature2\bar.h"
```

However, as this creates a dependency on the code generator folder structure, you should instead use one of the approaches described above.

More About

- “Files and Folders Created During Build Process” on page 19-57
- “Build and Run a Program” on page 25-36

Files and Folders Created During Build Process

In this section...

“Files Created During the Build Process” on page 19-57

“Folders Used by the Build Process” on page 19-62

Files Created During the Build Process

This section lists *model*. * files created during the code generation and build process for the GRT target when used with stand-alone models. Additional folders and files are created to support shared utilities and model references.

The build process derives many of the files from the model file you create with Simulink. You can think of the model file as a high-level programming language source file.

Note Files generated by the build process for an ERT based target are packaged slightly differently. Depending on model architectures and code generation options, the build process for a GRT based target might produce files not generated for an ERT based target.

Descriptions of the principal generated files follow. Within the generated file names cited as shown, *model* represents the name of the model for which you are generating code, and *subsystem* represents the name of a subsystem within the model.

- *model.rtw*

An ASCII file, representing the compiled model, generated by the build process. This file is analogous to the object file created from a high-level language source program. By default, the build process deletes this file when the build process is complete. However, you can choose to retain the file for inspection.

- *model.c*

C source code that corresponds to the model file. The Target Language Compiler generates this file. The file contains:

- Include files *model.h* and *model_private.h*
- Data except data placed in *model_data.c*

- Model-specific scheduler code
- Model-specific solver code
- Model registration code
- Algorithm code
- Optional GRT wrapper functions
- *model.h*

Defines model data structures and a public interface to the model entry points and data structures. Also provides an interface to the real-time model data structure (*model_rtM*) via access macros. *model.h* is included by subsystem *.c* files in the model. It includes:

- Exported Simulink data symbols
- Exported Stateflow machine parented data
- Model data structures, including *rtM*
- Model entry point functions
- *model_private.h*

Contains local `define` constants and local data required by the model and subsystems. This file is included by the generated source files in the model. You might need to include *model_private.h* when interfacing legacy hand-written code to a model. See “Dependencies When Interfacing Custom and Generated Code” on page 19-39 for more information. This header file contains:

- Imported Simulink data symbols
- Imported Stateflow machine parented data
- Stateflow entry points
- Simulink Coder details (various macros, `enums`, and so forth that are private to the code)
- *model_types.h*

Provides forward declarations for the real-time model data structure and the parameters data structure. These structure might be used by function declarations of reusable functions. *model_types.h* is included by the generated header files in the model.

- *model_data.c*

A conditionally generated C source code file containing declarations for the parameters data structure and the constant block I/O data structure, and zero representations for structure data types that are used in the model. If these data structures are not used in the model, *model_data.c* is not generated. Note that these structures are declared `extern` in *model.h*. When present, this file contains:

- Constant block I/O parameters
- Include files *model.h* and *model_private.h*
- Definitions for the zero representations for user-defined structure data types used by the model
- Constant parameters
- *model.exe* (Microsoft Windows platforms) or *model* (UNIX platforms), generated in the current folder, not in the build folder

Executable program file created under control of the `make` utility by your development system (unless you have explicitly specified that the code generator produce code only and skip the rest of the build process)

- *model.mk*

Customized makefile generated by the build process. This file builds an executable program file.

- *rtmodel.h*

Contains `#include` directives required by static main program modules such as *rt_main.c*. Since these modules are not created at code generation time, they include *rt_model.h* to access model-specific data structures and entry points. If you create your own main program module, take care to include *rtmodel.h*.

- *rtwtypes.h*

A header file that includes *simstruc_types.h* which, in turn, includes *tmwtypes.h*. For ERT based targets that do not generate a GRT interface and do not have noninlined S-functions, *rtwtypes.h* is a reduced file based on the model configuration. In this case, *rtwtypes.h* does not include *simstruc_types.h* and *tmwtypes.h*, but provides the essential type definitions, `#define` statements, and enumerations. For more information, see “Dependencies When Interfacing Custom and Generated Code” on page 19-39.

- *rt_defines.h*

Contains type definitions for special mathematical constants (such as π and e) and defines the `UNUSED_PARAMETER` macro. File is generated when a mathematical constant definition is required or when a model function argument is required but not accessed by the function body.

- `multiword_types.h`

Contains type definitions for wide data types and their chunks. File is generated when you use multiword data types or when you select one or more of the following in the Configuration Parameters dialog box:

- **All Parameters > MAT-file logging**
 - **Code Generation > Interface > External mode**
- `model_reference_types.h`

Contains type definitions for timing bridges. File is generated for a model reference target or a model containing model reference blocks.

- `builtin_typeid_types.h`

Defines an enumerated type corresponding to built-in data types. File is generated when your model contains a Stateflow chart that uses messages or when you select one or more of the following in the Configuration Parameters dialog box:

- **All Parameters > MAT-file logging**
 - Any C API option at **Code Generation > Interface**
- `rt_nonfinite.c`, `rtGetInf.c`, `rtGetNaN.c`

C source files that declare and initialize global nonfinite values for `inf`, `minus inf`, and `nan`. Nonfinite comparison functions are also provided. Conditions under which these files are generated include the following:

- The generated code requires nonfinite numbers.
 - You select **Configuration Parameters > All Parameters > MAT-file logging**.
 - The model contains S-functions.
 - You select `grt.tlc` for **Configuration Parameters > Code Generation > System target file** and **Configuration Parameters > All Parameters > Classic call interface**.
- `rt_nonfinite.h`, `rtGetInf.h`, `rtGetNaN.h`

C header files that define `extern` references to nonfinite variables and functions. Conditions under which these files are generated include the following:

- The generated code requires nonfinite numbers.
- You select **Configuration Parameters > All Parameters > MAT-file logging**.
- The model contains S-functions.
- You select `grt.tlc` for **Configuration Parameters > Code Generation > System target file** and **Configuration Parameters > All Parameters > Classic call interface**.
- `rtw_proj.tmw`

Marker file generated for the `make` utility to use to help determine when to rebuild objects.

- `model.bat`

Windows batch file used to set up the compiler environment and invoke the `make` command.

If you are using the toolchain approach for the build process, you also can use this batch file to extract information from the generated makefile, `model.mk`. The information includes macro definitions and values that appear in the makefile, such as `CFLAGS` (C compiler flags) and `CPP_FLAGS` (C++ compiler flags). With current working folder set to the folder containing `model.bat`, in the Command Window, type:

```
!model.bat info
```

On UNIX systems, the code generator does not create the `model.bat` file. To extract information for toolchain approach builds from the generated makefile on these systems, in the Command Window, type:

```
!gmake -f model.mk info
```

- `modelsources.txt`

List of additional sources that should be included in the compilation.

Optional files:

- `model_targ_data_map.m`

MATLAB language file used by external mode to initialize the external mode connection

- *model_dt.h*

C header file used for supporting external mode. Declares structures that contain data type and data type transition information for generated model data structures.

- *subsystem.c*

C source code for each noninlined nonvirtual subsystem or copy thereof when the subsystem is configured to place code in a separate file

- *subsystem.h*

C header file containing exported symbols for noninlined nonvirtual subsystems. Analogous to *model.h*.

- *model_capi.h*, *model_capi.c*

Header and sources file that contain data structures that describe the model signals, states, and parameters without using external mode. See “Exchange Data Between Generated and External Code Using C API” on page 28-2.

- *rt_sfcn_helper.h*, *rt_sfcn_helper.c*

Header and source files providing functions used by noninlined S-functions in a model. The functions *rt_CallSys*, *rt_enableSys*, and *rt_DisableSys* are used when noninlined S-functions call downstream function-call subsystems.

In addition, when you select the **Create code generation report** check box, the code generator produces a set of HTML files (one for each source file plus a *model_contents.html* index file) in the `html` subfolder within your build folder.

The described source files have dependency relationships, and there are additional file dependencies that you might need to consider. These dependencies are described in “Generated Source Files and File Dependencies” on page 19-38.

Folders Used by the Build Process

The code generator build process places output files in these folders:

- The “Code generation folder” (default is current working folder, `pwd`)

If you choose to generate an executable program file, the code generator writes the file *model.exe* (Windows) or *model* (UNIX) to your code generation folder. To choose a folder other than *pwd*, use Simulink preferences.

- The “Simulation cache folder” (default is current working folder, *pwd*)

If you choose to generate code for simulation, the code generator writes any model build artifacts for simulation to the simulation cache folder. To choose a folder other than *pwd*, use Simulink preferences.

- The build folder — *model_target_rtw*

A subfolder within your code generation folder. The build folder name is *model_target_rtw*, where *model* is the name of the source model and *target* is the selected target type (for example, *grt* for the generic real-time target). The build folder stores generated source code and other files created during the build process (except the executable program file).

- Reference model code and shared utilities folder — *slprj*

A subfolder within your code generation folder. When models referenced with Model blocks are built for simulation or code generation, files are placed in *slprj*. The Embedded Coder configuration has an option (*UtilityFuncGeneration*) that places generated shared code in *slprj* without the use of model reference. Subfolders in *slprj* provide separate places for simulation code, some generated code, utility code shared between models, and other files. File types of particular importance to developers of generated code are:

- Header files from models referenced by this model

slprj/target/model/referenced_model_includes

- Model reference code generation target files

slprj/target/model

- MAT-files used during code generation of model reference code generation target and standalone code generation

slprj/target/model/tmwinternal

- Shared utilities

slprj/target/_sharedutils

See “Work with Code Generation Folders” on page 8-13 for more information on organizing your files regarding code generation folders.

The build folder contains the generated code modules *model.c*, *model.h*, and the generated makefile *model.mk*.

Depending on the target, code generation, and build options you select, the build folder might also include

- *model.rtw*
- Object files (.obj or .o)
- Code modules generated from subsystems
- HTML summary reports of files generated (in the `html` subfolder)
- TLC profiler report files
- Block I/O, state, and parameter tuning information file (*model_capi.c*)

For additional information about using code generation folders, see “Code Generation Folder Structure for Model Reference Targets” on page 8-15 and “Customize Build to Use Shared Utility Code” on page 35-42.

More About

- “Generated Source Files and File Dependencies” on page 19-38
- “Enable Build Process When Folder Names Have Spaces” on page 19-65
- “Build and Run a Program” on page 25-36

Enable Build Process When Folder Names Have Spaces

The code generation software uses alternate folder name support—which is operating system specific—to process folder names that include spaces. On Windows systems, if the folder is a UNC location or if the path to this folder contains spaces, the code generation software maps a drive corresponding to the MATLAB installation folder. The build process provides similar support for other build-related folders. The Build Process Folder Support on Windows table provides a summary of support and limitations.

Spaces in folder names can appear in the paths to build-related locations:

- `matlabroot` – path to your MATLAB installation folder.

An example is a `matlabroot` similar to `C:\Program Files\MATLAB\R2015b`.

- `pwd` – current working folder from which you start the build.

An example is a `pwd` at the start of a build similar to `C:\Users\username\Documents\My Work`.

- The installation folder for a compiler used for the build.

If your work environment includes one or more of the preceding scenarios, use the following support mechanisms for the build process:

- If you are using the toolchain approach to build generated code, the toolchain manages spaces in folder names by using alternate names from the platform (for example, Windows or UNIX). For more information about how the toolchain `TransformPathsWithSpaces` method manages these names, see `addAttribute(coder.make.ToolchainInfo)`.
- If you are using the template makefile approach to build generated code, the template makefile (`.tmf`) requires code to manage spaces in folder names. When the alternate folder names (Windows short names) differ from the file system folder names (Windows long names), add this code to the makefile.

```
ALT_MATLAB_ROOT      = |>ALT_MATLAB_ROOT<|
ALT_MATLAB_BIN       = |>ALT_MATLAB_BIN<|
!if "$(MATLAB_ROOT)" != "$(ALT_MATLAB_ROOT)"
MATLAB_ROOT = $(ALT_MATLAB_ROOT)
!endif
!if "$(MATLAB_BIN)" != "$(ALT_MATLAB_BIN)"
MATLAB_BIN = $(ALT_MATLAB_BIN)
!endif
```

When the values of the location tokens are not equal, this code replaces `MATLAB_ROOT` with `ALT_MATLAB_ROOT`. The replacement indicates that the path to your MATLAB installation folder includes spaces. This code applies the same type of replacement for `MATLAB_BIN` with `ALT_MATLAB_BIN`. The preceding code is specific to `nmake`. For platform-specific examples, see the supplied template makefiles.

With either build approach, when there is an issue with support for creation of alternate names (short names), builds errors can occur on Windows. If a build generates an error message similar to the following message, see “Troubleshooting Errors When Folder Names Have Spaces” on page 19-68.

```
NMAKE : fatal error U1073: don't know how to make ' ...
```

Note: When using operating system commands, such as `system` or `dos`, enclose paths that specify executable files or command parameters in double quotes (" "). For example:

```
system('dir "D:\Applications\Common Files"')
```

Build Process Folder Support on Windows

Build Process Folders	Approach for Paths with UNC or Spaces	Support for Windows Platform
<p><code>matlabroot</code> folder</p> <p>Note: The <code>matlabroot</code> value is derived from the MATLAB installation location.</p>	<p>During a build, a UNC location such as:</p> <pre>\\networkdrive\matlab\R20xxb</pre> <p>could be remapped as:</p> <pre>T:\</pre> <p>During a build on a Windows system with short file name (8.3) support (default for Windows using NTFS), the build process uses the Windows API <code>getShortPathName()</code> for the folder location.</p>	<p>Build process folder support available independent of file system (NTFS or ReFS) or file system configuration for short file name support.</p> <p>Limitations:</p> <p>On systems that require drive mapping for the installation location, the build process requires that a drive letter is available for mapping.</p> <p>On systems without short file name (8.3) support (using ReFS or using NTFS with 8.3 support disabled), the final folder in the installation location</p>

Build Process Folders	Approach for Paths with UNC or Spaces	Support for Windows Platform
	<p>During a build on a Windows system without short file name (8.3) support (systems using ReFS or using NTFS with 8.3 support disabled), a location with spaces in the path such as:</p> <p>C:\Program Files\MATLAB\R20xxb</p> <p>could be remapped as:</p> <p>T:\R20xxb</p>	<p>may not contain spaces. For example, a final folder name:</p> <p>C:\Program Files\MATLAB\R20xxb update 2</p> <p>is not supported.</p>
<p>Code generation folder</p> <p>Simulation cache folder</p>	<p>For UNC locations, build process temporarily maps a drive by using the shell commands <code>pushd</code> and <code>popd</code>.</p>	<p>Build process folder support is available independent of file system (NTFS or ReFS) or file system configuration for short path name support.</p>
<p>Custom code source file locations—among others, these locations include those specified by:</p> <ul style="list-style-type: none"> • <code>rtwmakecfg.m</code> • Configuration Parameters <ul style="list-style-type: none"> > Code Generation > Custom Code > Additional build information • Code replacement library 	<p>For paths with spaces, build process uses the Windows short path name (8.3) by using the Windows API <code>getShortPathName()</code>.</p>	<p>Build process folder support depends on NTFS file system and requires Windows default support (registry sets <code>NtfsDisable8dot3NameCreation</code> to value of 2 or 0).</p> <p>Limitations: Build process does not support spaces in the path to these folders for:</p> <ul style="list-style-type: none"> • NTFS file system with short path name support disabled • ReFS file system (this file system does not support short path names)

Troubleshooting Errors When Folder Names Have Spaces

On Windows, when there is an issue with support for creation of short file names, build process errors can occur. When this issue affects a build, you see an error message similar to:

```
NMAKE : fatal error U1073: don't know how to make 'C:\Work\My'
```

This message could occur if a space in the folder name (`C:\Work\My Models`) prevents the build process from finding the model or a file to build. For descriptions of the build-related folders that are sensitive to a space in the folder name or path, see the Build Process Folder Support on Windows table in “Folders Used by the Build Process” on page 19-62.

Tip To avoid issues from folder names with spaces when Windows short file name support for file names is disabled, do not use paths with spaces.

For example, install any third-party software to paths without spaces. Do not use paths with spaces for folders containing your models, source files, or libraries.

An issue can occur with builds using folders with Windows long names (for example, folder names with spaces), because it is possible to disable Windows alternate name support. The build process uses this support on Windows systems. There are many terms for this file, folder, and path alternate name support:

- 8.3 name
- DOS path
- short file name (SFN, ShortFileName)
- long name alias
- Windows path alias

Check the type of file system used for the drive. In Windows Explorer, right-click the drive icon and select properties.

- If the file system is ReFS (Resilient File System), you have found the issue. The ReFS does not provide short file name support. Except for the MATLAB installation folder, the build process does not support folder names with spaces for the ReFS file system. If your work environment requires short file name support for the build folder or for additional custom code folders, do not use ReFS.

- If the file system is NTFS (New Technology File System), it is possible that the build error is related to a registry setting incompatibility. Continue with troubleshooting steps.

To check whether the error cause is an issue with short file name support on a system using NTFS, check the Windows registry setting that enables the creation of short names for files, folders, and paths.

- 1 From the Windows Start menu, type `cmd`, right-click on the `cmd.exe` icon, and select **Run as administrator**.
- 2 Change to the `windows\system32` folder and query the `NtfsDisable8dot3NameCreation` status by typing:

```
> fsutil 8dot3name query
```

- 3 If the registry state of `NtfsDisable8dot3NameCreation` is not 2, the default (Volume level setting), change the value to 2 by typing:

```
> fsutil 8dot3name set 2
```

For more information about enabling creation of short names. See <http://technet.microsoft.com/en-us/library/ff621566.aspx>.

Changing the registry setting enables only creation of short names for files and folders that are created after the change.

- 4 To create short names for files created while short name creation was disabled, use the `fsutil` utility at the Windows command line.

The syntax to set the short name is:

```
> fsutil file setshortname <FileName> <ShortName>
```

For example, to create the short name `PROGRA-1` for the long name `C:\Program Files`, type:

```
> fsutil file setshortname "C:\Program Files" PROGRA-1
```

The `C:\Program Files` folder name is in quotations because it has spaces.

- 5 To check that the short name was created, use the `dir` command with `/x` option to show short names.

```
> dir C:\ /x
```

See Also

`addAttribute` (`coder.make.ToolchainInfo`)

More About

- “Files and Folders Created During Build Process” on page 19-57
- “Build and Run a Program” on page 25-36

External Websites

- MATLAB Answers: “Why is the build process failing ...?”
- <http://technet.microsoft.com/en-us/library/cc788058.aspx>
- <http://technet.microsoft.com/en-us/library/cc788058.aspx>

How Code Is Generated from a Model

The build process begins with the Simulink software compiling the block diagram. During this stage, Simulink

- Evaluates simulation and block parameters
- Propagates signal widths and sample times
- Determines the execution order of blocks within the model
- Computes work vector sizes, such as those used by S-functions. (For more information about work vectors, see “DWork Vector Basics”).

Model Compilation

When Simulink completes the process of compiling the block diagram, it compiles a partial representation of the model. This description is stored in a language-independent format in the ASCII file *model.rtw*. The *model.rtw* file is the input to the next stage of the build process. For a general description of the *model.rtw* file format, see “model.rtw File and Scopes”.

Code Generation

The code generator uses the Target Language Compiler (TLC) and a supporting TLC function library to transform the partial model description stored in *model.rtw* into target-specific code.

The target language executed by the TLC is an interpreted programming language designed to convert a model description to code. The TLC executes a TLC program comprising several target files (*.tlc* scripts). The TLC scripts specify how to generate code from the model, using the *model.rtw* file as input.

The TLC

- 1 Reads the *model.rtw* file
- 2 Interprets and executes commands in a system target file

The system target file is the entry point or main file. You select it from the target files that are available on the MATLAB path with the system target file browser or you can type the name of such a file on your system before building.

3 Interprets and executes commands in block-level target files

For blocks in the Simulink model, there is a corresponding target file that is either dynamically generated or statically provided.

Note The code generator executes user-written S-function target files, but optionally executes block target files for Simulink blocks.

4 Writes a source code version of the Simulink block diagram

More About

- “Files and Folders Created During Build Process” on page 19-57
- “Build and Run a Program” on page 25-36

Code Generation of Matrices and Arrays

In this section...

“Code Generator Matrix Parameters” on page 19-74

“Internal Data Storage for Complex Number Arrays” on page 19-76

MATLAB, Simulink, and the code generator store matrix data and arrays (1-D, 2-D, ...) in column-major format as a vector. Column-major format orders elements in a matrix starting from the first column, top to bottom, and then moving on to the next column. For example, the following 3x3 matrix:

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

translates to an array of length 9 in the following order:

```
A(1) = A(1,1) = 1;
A(2) = A(2,1) = 4;
A(3) = A(3,1) = 7;
A(4) = A(1,2) = 2;
A(5) = A(2,2) = 5;
and so on.
```

In column-major format, the next element of an array in memory is accessed by incrementing the first index of the array. For example, these element pairs are stored sequentially in memory:

- $A(i)$ and $A(i+1)$
- $B(i, j)$ and $B(i+1, j)$
- $C(i, j, k)$ and $C(i+1, j, k)$

For more information on the internal representation of MATLAB data, see “MATLAB Data” in the MATLAB External Interfaces document.

Code generation software uses column-major format for several reasons:

- The world of signal and array processing is largely column major: MATLAB, LAPack, Fortran90, DSP libraries.

- A column is equivalent to a channel in frame-based processing. In this case, column-major storage is more efficient.
- A column-major array is self-consistent with its component submatrices:
 - A column-major 2-D array is a simple concatenation of 1-D arrays.
 - A column-major 3-D array is a simple concatenation of 2-D arrays.
 - The stride is the number of memory locations to index to the next element in the same dimension. The stride of the first dimension is 1 element. The stride of the n th dimension element is the product of sizes of the lower dimensions.
 - Row-major n -D arrays have their stride of 1 for the highest dimension. Submatrix manipulations are typically accessing a scattered data set in memory, which does not allow for efficient indexing.

C typically uses row-major format. MATLAB and Simulink use column-major format. You cannot configure the code generation software to generate code with row-major ordering. If you are integrating legacy C code with the generated code, consider the following:

To	Consider
Integrate row-major data in legacy C code with Simulink generated functions.	<ul style="list-style-type: none"> • Transposing the row-major data in your legacy C code into column-major format as a 1-D array. • Using a “Access Data Through Functions with Custom Storage Class GetSet”. The <code>GetSet</code> custom storage class replaces reads from (<code>get</code>) and writes to (<code>set</code>) with user-specified <code>GetSet</code> functions. The user-specified <code>GetSet</code> function has a single index argument, requiring the function to process the argument and access the target row and column data element. Using the <code>GetSet</code> function impacts code efficiency.
Integrate legacy C code functions requiring row-major data with Simulink generated data.	Using the Legacy Code Tool option <code>convert2DMatrixToRowMajor</code> to create an S-function that integrates legacy code functions with Simulink generated data. See <code>legacy_code</code> . The generated code stores the data as a 1-D array. Using <code>convert2DMatrixToRowMajor</code> impacts code efficiency.

Code Generator Matrix Parameters

The compiled model file, *model.rtw*, represents matrices as character vectors in MATLAB syntax, without an implied storage format. This format allows you to copy the character vector out of an *.rtw* file, paste it into a MATLAB file, and have it recognized by MATLAB.

The code generator declares Simulink block matrix parameters as scalar or 1-D array variables

```
real_T scalar;
real_T mat[ nRows * nCols ];
```

real_T can be a data type supported by Simulink. It matches the variable type given in the model file.

For example, the 3-by-3 matrix in the 2-D Look-Up Table block

```
 1  2  3
 4  5  6
 7  8  9
```

is stored in *model.rtw* as

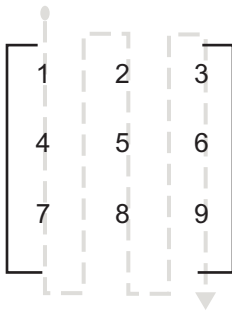
```
Parameter {
  Name  "OutputValues"
  Value  Matrix(3,3)
  [[1.0, 2.0, 3.0]; [4.0, 5.0, 6.0]; [7.0, 8.0, 9.0]];
  String  "t"
  StringType  "Variable"
  ASTNode {
    IsNonTerminal  0
    Op  SL_NOT_INLINED
    ModelParameterIdx  3
  }
}
```

and results in this definition in *model.h*

```
typedef struct Parameters_tag {
  real_T s1_Look_Up_Table_2_D_Table[9];
  /* Variable:s1_Look_Up_Table_2_D_Table
   * External Mode Tunable:yes
   * Referenced by block:
   * <S1>/Look-Up Table (2-D
   */
```

```
[ ... other parameter definitions ... ]
} Parameters;
```

The `model.h` file declares the actual storage for the matrix parameter. You can see that the format is column-major.



```
Parameters model_P = {
  /* 3 x 3 matrix s1_Look_Up_Table_2_D_Table */
  { 1.0, 4.0, 7.0, 2.0, 5.0, 8.0, 3.0, 6.0, 9.0 },
  [ ... other parameter declarations ... ]
};
```

Internal Data Storage for Complex Number Arrays

Simulink and code generator internal data storage formatting differs from MATLAB internal data storage formatting only in the storage of complex number arrays. In MATLAB, the real and imaginary parts are stored in separate arrays. In Simulink and the code generator, the parts are stored in an "interleaved" format. The numbers in memory alternate real, imaginary, real, imaginary, and so forth. This convention allows efficient implementations of small signals on Simulink lines, for Mux blocks, and other "virtual" signal manipulation blocks. For example, the signals do not actively copy their inputs, just the references.

More About

- "How Code Is Generated from a Model" on page 19-71

Generated Code Considerations

Requirements for Signed Integer Representation

You must compile the code produced by the code generator for target hardware by using a two's complement representation for signed integer values. The code generator does not provide a check for this configuration in the generated code.

Default Folder for Code Generation

When your default folder is set to the root folder of a drive, such as **C:**, the code generator cannot generate code for your model.

More About

- “How Code Is Generated from a Model” on page 19-71

About Shared Utility Code

Blocks in a model can require common functionality to implement their algorithm. Often, it is most efficient to modularize this functionality into standalone support or helper functions, rather than inlining the code for the functionality for each block instance.

Typically, functions that can have multiple callers are packaged into a library. Traditionally, such functions are defined statically, that is, the function source code exists in a file before you use the code generator to produce code for your model.

In other cases, several model- and block-specific properties specify which functions are used and their behavior. Also, these properties determine type definitions (for example, `typedef`) in shared utility header files. Since there are many possible combinations of properties that determine unique behavior, it is not practical to define statically all possible function files before code generation. Instead, use the shared utility mechanism, which produces support functions during code generation.

More About

- “Controlling Shared Utility Code Placement” on page 19-79
- “`rtwtypes.h` and Shared Utility Code” on page 19-81
- “Incremental Shared Utility Code Generation and Compilation” on page 19-82
- “Shared Utility Checksum” on page 19-83
- “Shared Fixed-Point Utility Functions” on page 19-88
- “Share User-Defined Data Types Across Models” on page 19-90
- “Cross-Release Shared Code Reuse” on page 19-95

Controlling Shared Utility Code Placement

You control the shared utility code placement mechanism with the **Shared code placement** option on the **Code Generation > Interface** pane in the Configuration Parameters dialog box. By default, the option is set to **Auto**. For this setting, if the model you are building does not include Model blocks, the code generator places code required for fixed-point and other utilities in one of the following:

- The *model.c* or *model.cpp* file
- A separate file in the build folder (for example, *vdp_grt_rtw*)

The code is specific to the model.

If a model does contain Model blocks, the code generator creates and uses a shared utilities folder within `slprj`. Model reference builds require the use of shared utilities. The naming convention for shared utility folders is `slprj/target/_sharedutils`. *target* is `sim` for simulations with Model blocks or the name of the system target file for code generation.

```
slprj/sim/_sharedutils      % folder used with simulation
slprj/grt/_sharedutils     % folder used with grt.tlc STF
slprj/ert/_sharedutils     % folder used with ert.tlc STF
slprj/mytarget/_sharedutils % folder used with mytarget.tlc STF
```

To force a model build to use the `slprj` folder for shared utility generation, even when the current model does not contain Model blocks, set the **Shared code placement** option to **Shared location**. The code generator places utilities under the `slprj` folder rather than in the normal build folder. This setting is useful when you are manually combining code from several models, as it prevents symbol collisions between the models.

Avoid Duplicate Header Files for Exported Data

If you control the file placement of declarations for signals, parameters, and states by applying storage classes and custom storage classes, and if the code generator places utility code in a shared location, the exported header files appear in the shared utility folder. For example, you can specify a header file for a piece of data through:

- The **Code Generation** tab in a Signal Properties dialog box.
- The `HeaderFile` property of a data object. Data objects are objects of the classes `Simulink.Signal` and `Simulink.Parameter`.

If you want the declaration to appear in the file *model.h*, it is a best practice to leave the header file name unspecified. By default, the code generator places data declarations in *model.h*.

If you specify *model.h* as the header file name, and if the code generator places utility code in a shared location, you cannot generate code from the model. The code generator cannot create the file *model.h* in both the model build folder and the shared utility folder.

More About

- “About Shared Utility Code” on page 19-78
- “rtwtypes.h and Shared Utility Code” on page 19-81

rtwtypes.h and Shared Utility Code

The generated header file `rtwtypes.h` provides fundamental type definitions, `#define` statements, and enumerations. The location of this file is controlled by whether the build process is using the shared utilities folder. If a shared folder is used, the build process places `rtwtypes.h` in `slprj/target/_sharedutils`. Otherwise, the software places `rtwtypes.h` in `model_target_rtw`.

Adding a model to a model hierarchy or changing an existing model in the hierarchy can result in updates to the shared `rtwtypes.h` file during code generation. If updates occur, recompile and, depending on your development process, reverify previously generated code. To minimize updates to the `rtwtypes.h` file, make the following changes in the Configuration Parameters dialog box:

- On the **Interface** pane, select **Support: complex numbers** even if the model does not currently use complex data types. Selecting this option protects against a future need to add support for complex data types when integrating code.
- On the **All Parameters** tab, clear **Support: non-inlined S-functions**. If you use non-inlined S-functions in the model, this option generates an error.
- On the **All Parameters** tab, clear **Classic call interface**. Disables use of the GRT interface.

More About

- “About Shared Utility Code” on page 19-78
- “Controlling Shared Utility Code Placement” on page 19-79
- “rtwtypes.h and Shared Utility Code” on page 19-81

Incremental Shared Utility Code Generation and Compilation

As explained in “Controlling Shared Utility Code Placement” on page 19-79, you can specify that C source files, which contain function definitions, and header files, which contain macro definitions, be generated in a shared utilities folder. For this discussion, the term functions means functions and macros.

A shared function can be used by blocks within the same model and by blocks in different models when using model reference or when building multiple standalone models from the same start build folder. However, the code generator produces the code for a given function only once for the block that first triggers code generation. As the product determines the need to generate function code for subsequent blocks, it performs a file existence check. If the file exists, the function is not regenerated. Thus, the shared utility code mechanism requires that a given function and file name represent the same functional behavior regardless of which block or model generates the function. To satisfy this requirement:

- Model properties that determine function behavior are included in a shared utility checksum or determine the function and file name.
- Block properties that determine the function behavior also determine the function and file name.

During compilation, makefile rules for the shared utilities folder are configured to compile only new C files, and incrementally archive the object file into the shared utility library, `rtwshared.lib` or `rtwshared.a`. Thus, incremental compilation is also done.

More About

- “About Shared Utility Code” on page 19-78
- “Incremental Shared Utility Code Generation and Compilation” on page 19-82
- “Shared Utility Checksum” on page 19-83

Shared Utility Checksum

In this section...

“About the Shared Utility Checksum” on page 19-83

“View the Shared Utility Checksum Hash Table” on page 19-83

“Relate the Shared Utility Checksum to Configuration Parameters” on page 19-85

About the Shared Utility Checksum

When you explicitly configure a model to use the shared location or the model contains Model blocks, the code generator uses the shared utilities folder. For more information, see “Incremental Shared Utility Code Generation and Compilation” on page 19-82. During code generation, if the configuration file `slprj/target/_sharedutils/checksummap.mat` exists relative to the current folder, the code generator reads that file. The code generator verifies that the current model that you are building has configuration properties that match the checksum of properties from the shared utility model.

If mismatches occur between the properties defined in `checksummap.mat` and the current model properties, you see an error. Use the error message to diagnose and resolve the issue.

View the Shared Utility Checksum Hash Table

It is helpful to view the property values that contribute to the checksum. This example uses the `rtwdemo_lct_start_term.slx` model. To load the `checksum.mat` file into MATLAB and view the `targetInfoStruct` that defines the checksum-related properties:

- 1 Open the `rtwdemo_lct_start_term.slx` model. In the Command Window, type:
`rtwdemo_lct_start_term`
- 2 Create and move to a new working folder.
`mkdir C:\Temp\demo`
`cd C:\Temp\demo`
- 3 Save a copy of the model in the folder.
- 4 Build the model by using the **Simulink > Code > C/C++ Code > Build Model** command. This model is already set up to produce shared utilities.

- 5 Move to the `_sharedutils` folder created by the build process.

```
cd C:\Temp\demo\slprj\grt\_sharedutils
```

- 6 Load the `checksummap.mat` file into MATLAB.

```
load checksummap
```

- 7 Display the contents of `hashTbl.targetInfoStruct` and examine the checksum-related property values.

```
hashTbl.targetInfoStruct
```

For this example, the Command Window displays `hashTbl.targetInfoStruct` for the shared utilities that you generated from the model:

```

    ShiftRightIntArith: 'on'
    ProdShiftRightIntArith: 'on'
        Endianness: 'LittleEndian'
        ProdEndianness: 'LittleEndian'
        wordlengths: '8,16,32,32,64,32,64,64,64,64'
        Prodwordlengths: '8,16,32,32,64,32,64,64,64,64'
        TargetWordSize: '64'
        ProdWordSize: '64'
    TargetHWDeviceType: 'Custom Processor->MATLAB Host Computer'
    ProdHWDeviceType: 'Intel->x86-64 (Windows64)'
    TargetIntDivRoundTo: 'Zero'
    ProdIntDivRoundTo: 'Zero'
        tmfName: 'grt_vcx64.tmf'
        toolchainName: ''
            computer: 'PCWIN64'
    UseDivisionForNetSlopeComputation: 'off'
    PurelyIntegerCode: 'off'
    PortableWordSizes: 'off'
    SupportNonInlinedSFcns: ''
        TargetLibSuffix: ''
    RTWReplacementTypes: ''
        MaxIdInt8: 'MAX_int8_T'
        MinIdInt8: 'MIN_int8_T'
        MaxIdUInt8: 'MAX_uint8_T'
        MaxIdInt16: 'MAX_int16_T'
        MinIdInt16: 'MIN_int16_T'
        MaxIdUInt16: 'MAX_uint16_T'
        MaxIdInt32: 'MAX_int32_T'
        MinIdInt32: 'MIN_int32_T'
        MaxIdUInt32: 'MAX_uint32_T'
        BooleanTrueId: 'true'
        BooleanFalseId: 'false'
    TypeLimitIdReplacementHeaderFile: ''
        SharedCodeRepository: ''
            TargetLang: 'C'
    PreserveExternInFcnDecls: 'on'
    EnableSignedRightShifts: 'on'
    EnableSignedLeftShifts: 'on'

```



```

        TflName: 'None'
        TflChecksum: [1.6615e+09 521991164 2.2147e+09 1.7704e+09]
        UtilMemSecName: 'Default'
        CodeCoverageChecksum: [3.6498e+09 78774415 2.5508e+09 2.1183e+09]
        TargetLargestAtomicInteger: 'Char'
        TargetLargestAtomicFloat: 'None'
        ProdLargestAtomicInteger: 'Char'
        ProdLargestAtomicFloat: 'Float'
        LongLongMode: 'on'
        ProdLongLongMode: 'off'
        CollapseNonTrivialExpressions: 'false'

```

Relate the Shared Utility Checksum to Configuration Parameters

Examine the `targetInfoStruct` hash table from the shared utility model. Some key-value pairs relate directly to a model property. Other pairs relate to groups of properties.

The following table describes the key-value pairs.

Key Names	Model Properties
ShiftRightIntArith	TargetShiftRightIntArith
ProdShiftRightIntArith	ProdShiftRightIntArith
Endianess	TargetEndianess
ProdEndianess	ProdEndianess
wordlengths	TargetBitPerChar, TargetBitPerShort, TargetBitPerInt, TargetBitPerLong, TargetBitPerLongLong, TargetBitPerFloat, TargetBitPerDouble, TargetBitPerPointer
Prodwordlengths	ProdBitPerChar, ProdBitPerShort, ProdBitPerInt, ProdBitPerLong, ProdBitPerLongLong, ProdBitPerFloat, ProdBitPerDouble, ProdBitPerPointer
TargetWordSize	TargetWordSize
ProdWordSize	ProdWordSize
TargetHWDeviceType	TargetHWDeviceType

Key Names	Model Properties
ProdHWDeviceType	ProdHWDeviceType
TargetIntDivRoundTo	TargetIntDivRoundTo
ProdIntDivRoundTo	ProdIntDivRoundTo
tmfName	TemplateMakefile
toolchainName	Toolchain
computer	return value of the computer command
UseDivisionForNetSlopeComputation	UseDivisionForNetSlopeComputation
PurelyIntegerCode	PurelyIntegerCode
PortableWordSizes	PortableWordSizes
SupportNonInlinedSFcns	SupportNonInlinedSFcns
TargetLibSuffix	Removed (not used)
RTWReplacementTypes	EnableUserReplacementTypes, ReplacementTypes
MaxIdInt8	MaxIdInt8
MinIdInt8	MinIdInt8
MaxIdUInt8	MaxIdUInt8
MaxIdInt16	MaxIdInt16
MinIdInt16	MinIdInt16
MaxIdUInt16	MaxIdUInt16
MaxIdInt32	MaxIdInt32
MinIdInt32	MinIdInt32
MaxIdUInt32	MaxIdUInt32
BooleanTrueId	BooleanTrueId
BooleanFalseId	BooleanFalseId
TypeLimitIdReplacementHeaderFile	TypeLimitIdReplacementHeaderFile
SharedCodeRepository	reserved (internal use only)
TargetLang	TargetLang
PreserveExternInFcnDecls	PreserveExternInFcnDecls

Key Names	Model Properties
EnableSignedRightShifts	EnableSignedRightShifts
EnableSignedLeftShifts	EnableSignedLeftShifts
TflName	CodeReplacementLibrary
TflChecksum	reserved (internal use only)
UtilMemSecName	MemSecFuncSharedUtil, MemSecPackage
CodeCoverageChecksum	reserved (internal use only)
TargetLargestAtomicInteger	TargetLargestAtomicInteger
TargetLargestAtomicFloat	TargetLargestAtomicFloat
ProdLargestAtomicInteger	ProdLargestAtomicInteger
ProdLargestAtomicFloat	ProdLargestAtomicFloat
LongLongMode	TargetLongLongMode
ProdLongLongMode	ProdLongLongMode
CollapseNonTrivialExpressions	reserved (internal use only)

More About

- “About Shared Utility Code” on page 19-78
- “Incremental Shared Utility Code Generation and Compilation” on page 19-82

Shared Fixed-Point Utility Functions

An important set of generated functions that are placed in the shared utility folder are the fixed-point support functions. Based on model and block properties, there are many possible versions of fixed-point utilities functions that make it impractical to provide a complete set as static files. Generating only the required fixed-point utility functions during the code generation process is an efficient alternative.

The shared utility checksum mechanism makes sure that several critical properties are identical for models that use the shared utilities. For the fixed-point functions, there are additional properties that determine function behavior. These properties are coded into the functions and file names to maintain requirements. The additional properties include:

Category	Function/Property
Block properties	<ul style="list-style-type: none"> Fixed-point operation being performed by the block Fixed-point data type and scaling (Slope, Bias) of function inputs and outputs Overflow handling mode (Saturation, Wrap) Rounding Mode (Floor, Ceil, Zero)
Model properties	<code>get_param(bdroot, 'NoFixptDivByZeroProtection')</code>

The naming convention for the fixed-point utilities is based on the properties as follows:

```
operation + [zero protection] + output data type + output bits +
[input1 data] + input1 bits + [input2 data type + input2 bits] +
[shift direction] + [saturate mode] + [round mode]
```

The file names shown are examples of generated fixed-point utility files. The function or macro names in the file are identical to the file name without the extension.

```
FIX2FIX_U12_U16.c
FIX2FIX_S9_S9_SR99.c
ACCUM_POS_S30_S30.h
MUL_S30_S30_S16.h
div_nzp_s16s32_floor.c
div_s32_sat_floor.c
```

For these examples, the respective fields correspond as follows:

Operation	FIX2FIX	FIX2FIX	ACCUM_POS	MUL	div	div
Zero protection	NULL	NULL	NULL	NULL	_nzp	NULL
Output data type	_U	_S	_S	_S	_s	_s
Output bits	12	9	30	30	16	32
Input data type	_U	_S	_S	_S [and _S]	s	NULL
Input bits	16	9	30	30 [and 16]	32	NULL
Shift direction	NULL	SR99	NULL	NULL	NULL	NULL
Saturate mode	NULL	NULL	NULL	NULL	NULL	_sat
Round mode	NULL	NULL	NULL	NULL	_floor	_floor

Note: For the ACCUM_POS example, the output variable is also used as one of the input variables. Therefore, only the output and second input is contained in the file and macro name. For the second div example, both inputs and the output have identical data type and bits. Therefore, only the output is included in the file and function name.

More About

- “About Shared Utility Code” on page 19-78

Share User-Defined Data Types Across Models

By default, the code generator produces duplicate type definitions in the `model_types.h` file for each model when you share user-defined data types between models. The code generator lets you generate custom-defined data type definitions into a header file in the `_sharedutils` folder that you can share across multiple models, removing the need for duplicate copies of the data type definitions.

In this section...

“About Sharing Data Types” on page 19-90

“Example: Sharing Simulink Data Type Objects” on page 19-91

“Example: Sharing Enumerated Data Types” on page 19-92

About Sharing Data Types

User-defined data types that you can set up in a shared header file include:

- Simulink data type objects that you instantiate from the classes `Simulink.AliasType`, `Simulink.Bus`, and `Simulink.NumericType`.
- Enumeration types that you define in MATLAB code

The general procedure for setting up user-defined data types to be shared among multiple models is as follows:

- 1 Define the data type.
- 2 Set data scope and header file properties that allow the data type to be shared.
- 3 Use the data type as a signal type in your models.
- 4 Before generating code for each model, set the **Shared code placement** parameter on the **Code Generation > Interface** pane of the Configuration Parameters dialog box to the value `Shared location`.
- 5 Generate code.

Note: You can configure the definition of the user-defined data type to occur in a header file that is located in the `_sharedutils` folder, however user-defined data type names are not used by the shared utility functions that are generated into the `_sharedutils` folder. Currently, the user-defined data type names are used only by model code located in code folders for each model.

For more information, see “Example: Sharing Simulink Data Type Objects” on page 19-91 and “Example: Sharing Enumerated Data Types” on page 19-92.

Example: Sharing Simulink Data Type Objects

To export a user-defined Simulink data type object for sharing among multiple models.

1 In the base workspace, create a data type object of one of the following classes:

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

For example:

```
a = Simulink.AliasType
```

2 To specify that the data type should be exported to a specified header file, use the data type object property `DataScope`. Specify the value 'Exported' for the `DataScope` property. For example:

```
a.DataScope = 'Exported'
```

3 To specify the name of the header file to which the data type should be exported, use the data type object property `HeaderFile`. For example:

```
a.HeaderFile = 'a.h'
```

Alternatively, if you leave the `HeaderFile` value empty, the name of the generated header file defaults to the name of the data type, in this case, `a.h`.

4 Use the data type object as a signal type in a model.

5 Before generating code, if you want to generate the header file into the shared utilities folder for sharing definitions between multiple models, set the **Shared code placement** parameter on the **Code Generation > Interface** pane of the Configuration Parameters dialog box to the value `Shared location`.

6 Generate code. Here is an example of the definition code that is generated to `a.h` in the shared utilities folder:

```
#ifndef RTW_HEADER_a_h_
#define RTW_HEADER_a_h_
#include "rtwtypes.h"
```

```
typedef double a;  
  
#endif      /* RTW_HEADER_a_h_ */
```

To share the data type definition from other models, the alternatives include:

- Simply use the same workspace variable in multiple models. If the contents of the data type object are the same, generating code does not regenerate the header file. For example, model A and model B can export the same data type to the same header file in the shared utilities folder.
- Define the same data type object in other models, but set it up to import the shared data type definition from the header file. Specify the value 'Imported' for the data type object property `DataScope`. For example:

```
a = Simulink.AliasType  
a.DataScope = 'Imported'  
a.HeaderFile = 'a.h'
```

Example: Sharing Enumerated Data Types

To export a user-defined enumerated data type for sharing among multiple models:

- 1 Define an enumerated data type in MATLAB code. For example, the following MATLAB file defines `BasicColors`:

```
% BasicColors.m  
classdef BasicColors < Simulink.IntEnumType  
    enumeration  
        Red(0)  
        Yellow(1)  
        Blue(2)  
    end  
    methods (Static = true)  
        function retVal = getDefaultvalue()  
            retVal = BasicColors.Blue;  
        end  
    end  
end
```

- 2 To specify that the data type should be exported to a specified header file, override the default `getDataScope` method of the enumerated class. Specify the value 'Exported' as the `getDataScope` return value. For example:


```

methods (Static = true)
...
    function retVal = getDataScope()
        retVal = 'Exported';
    end
...
end

```

- 3** To specify the name of the header file to which the data type should be exported, override the default `getHeaderFile` method of the enumerated class. Specify a file name as the `getHeaderFile` return value. For example:

```

methods (Static = true)
...
    function retVal = getHeaderFile()
        retVal = 'BasicColors.h';
    end
...
end

```

Alternatively, if you leave the return value of the `getHeaderFile` method unspecified, the name of the generated header file defaults to the name of the data type, in this case, `BasicColors.h`.

- 4** Here is `BasicColors.m` after the data scope and header file changes.

```

% BasicColors.m
classdef BasicColors < Simulink.IntEnumType
    enumeration
        Red(0)
        Yellow(1)
        Blue(2)
    end
    methods (Static = true)
        function retVal = getDefaultvalue()
            retVal = BasicColors.Blue;
        end
        function retVal = getDataScope()
            retVal = 'Exported';
        end
        function retVal = getHeaderFile()
            retVal = 'BasicColors.h';
        end
    end
end

```

```
end
```

Make sure `BasicColors.m` is present in the MATLAB path.

- 5 Use the type `enum:BasicColors` as a signal type in a model.
- 6 Before generating code, if you want to generate the header file into the shared utilities folder for sharing definitions between multiple models, set the **Shared code placement** parameter on the **Code Generation > Interface** pane of the Configuration Parameters dialog box to the value `Shared location`.
- 7 Generate code. Here is an example of the definition code that is generated to `BasicColors.h` in the shared utilities folder:

```
#ifndef RTW_HEADER_BasicColors_h
#define RTW_HEADER_BasicColors_h
/* Type definition for enum:BasicColors type */
typedef enum {
    Red = 0,
    Yellow,
    Blue
} BasicColors;
#endif
```

To share the data type definition from other models, the alternatives include:

- Simply use the same enumerated type as a signal type in multiple models. If the contents of the enumerated type are the same, generating code does not regenerate the header file. For example, model A and model B can export the same data type to the same header file in the shared utilities folder.
- Define the same enumerated type in other models, but set it up to import the shared data type definition from the header file. In the enumerated type definition file, specify the value `'Imported'` as the `getDataScope` return value. For example:

```
methods (Static = true)
...
    function retVal = getDataScope()
        retVal = 'Imported';
    end
...
end
```

More About

- “About Shared Utility Code” on page 19-78

Cross-Release Shared Code Reuse

In this section...

“Workflow for Shared Code Reuse” on page 19-95

“Shared Code Files That Require Editing” on page 19-96

When you generate code for a model, the code generator by default creates shared utility files that the model requires. When you generate code with different releases, the code generators can produce functionally identical shared files that contain some nonfunctional differences. For example, different comments and different coding style. When you use the same release to generate code for different models in different folders, you can also produce shared files with nonfunctional differences. For example, if you specify different `ParenthesesLevel` or `ExpressionFolding` values for the models, the code generator can produce shared files that contain different comments or different coding styles.

Integrated code that includes functionally identical shared files:

- Is more expensive to verify because each shared file requires verification.
- Produces compilation errors if the shared files define duplicate symbols.

To avoid these issues, you can specify the reuse of shared code from an existing folder, for example, a read-only library of verified code. In this case, the code generator does not create new shared utility files. The build process uses custom code or previously generated shared utility code from the folder. An administrator maintains and updates the read-only library.

Workflow for Shared Code Reuse

- 1 In the **Configuration Parameters > All Parameters > Existing shared code** field, enter the full path to your shared code folder.
- 2 Verify that the **Configuration Parameters > All Parameters > Use only existing shared code** diagnostic is set to **error** (default).
- 3 Remove the `slprj` folder or move to a new working folder.
- 4 Build your model. If you do not see an error, your shared code folder contains the required shared utility files.
- 5 If files are missing from the shared code folder, you see an error. To continue code generation with a locally generated version of the missing shared utility files:

- a Set **Configuration Parameters > All Parameters > Use only existing shared code** to warning.
- b Rebuild your model. The code generation process uses a locally generated version of the missing shared utility files.
- c Provide the administrator of the verified code library with your model and information about missing shared utility files. With the model, the administrator generates the required shared utility files. Using `sharedCodeUpdate`, the administrator adds the files to the shared code folder.
- d When the files are available in the shared code folder, repeat steps 1–4.

If the shared utility code is generated from library subsystems that are shared across models on page 6-28, you cannot reuse the code *across* releases because the code is release-specific—the symbol name and file name mangling includes the release number. The administrator must add the shared utility code generated for each release to the shared code folder.

The `sharedCodeUpdate` function can add files to the shared code folder that have identical content but different file and function names. This behavior is useful when you have different model components that require their own shared utility functions. Although some code is duplicated, the different model components can access the shared utility functions with which they were verified. To force model components to have their own versions of shared utility functions, configure naming rules to insert the model name into shared utility identifiers.

Shared Code Files That Require Editing

For most shared code files, you can specify master copies that you can reuse across releases without making changes. With some files, for example, `rtwtypes.h`, and `zero_crossing_types.h`, there are situations where manual editing is required to produce master copies that you can use with generated code from different releases. For example:

- The `rtwtypes.h` file generated by R2010a contains a checksum.

```
/* This ID is used to detect inclusion of an incompatible rtwtypes.h */  
#define RTWTYPES_ID_C08S16I32L64N64F0
```

For each R2010a version of `rtwtypes.h` that you want to include in your integration, copy the corresponding `#define` statement into your master copy of `rtwtypes.h`.

- In R2015a, the zero-crossing definitions moved from `rtwtypes.h` into `zero_crossing_types.h`. To create an `rtwtypes.h` file that is compatible with generated model code from different releases, in your master copy of `rtwtypes.h`, insert this statement.

```
#include "zero_crossing_types.h"  
Remove definitions from rtwtypes.h that zero_crossing_types.h provides.
```

See Also

`crossReleaseImport` | `crossReleaseExport` | `sharedCodeUpdate`

Related Examples

- “Cross-Release Code Integration”

Cross-Release Code Integration

In this section...
“Workflow” on page 19-98
“Limitations” on page 19-101
“Incorporate Model Reference Code” on page 19-102
“Simulink.Bus Support” on page 19-102
“Parameter Tuning” on page 19-104

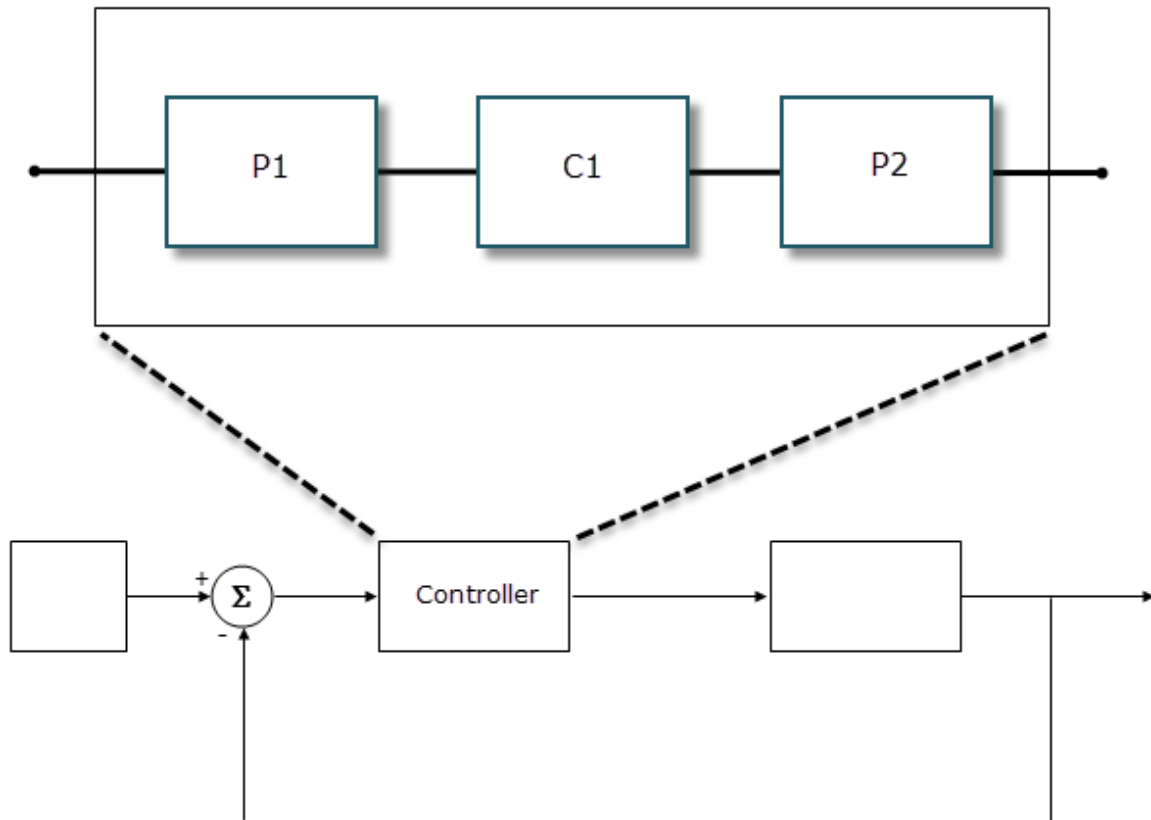
If you have an Embedded Coder license, you can integrate generated C code from previous releases (R2010a and later) with generated code from the current release when:

- The source models are single-rate.
- The source models are set to generate nonreusable code with function prototype control (root-level Inport and Outport blocks are mapped to step function arguments).
- The generated C code is from top-model and subsystem build processes.

When you reuse existing code without modification, you avoid the cost of reverification.

Workflow

Consider this control system model.



The Controller Model block references a model that consists of three components:

- P1 is a Model block, which references a model developed with a previous release, for example, R2015b. The generated model code, with the standalone code interface, is in the folder P1_ert_rtw.
- C1 is a Model block, which references a model that you are developing in the current release.
- P2 is a subsystem block developed with a previous release, for example, R2016a. The generated subsystem code is in the folder P2_ert_rtw.

To integrate code from previous releases with generated code from the current release, use this workflow:

1 Export components from previous releases

- a Add the `crossReleaseExport` function to the search paths for previous releases. For example, in the Command Window of a previous release, run this command:

```
addpath(fullfile(matlabRootCR, 'toolbox', 'coder', 'xrelexport'));  
matlabRootCR is the matlabroot value for your current release.
```

- b Using the previous releases, export generated component code. For example:

- From R2015b, run:

```
crossReleaseExport(fullfile(folderPath, 'P1_ert_rtw'))
```

- From R2016a, run:

```
crossReleaseExport(fullfile(folderPath, 'P2_ert_rtw'))
```

The `crossReleaseExport` function creates export artifacts in new folders `P1_R2015b` and `P2_R2016a`.

2 Specify an existing shared code folder

You can specify the reuse of shared code from an existing folder, for example, a library of verified code that an administrator maintains and updates. Specify the shared code folder for model components that require code generation in the current release, for example, `Controller` and `C1`.

In the **Configuration Parameters > All Parameters > Existing shared code** field, enter the full path to the shared code folder.

3 Import components into current release

From the current release, using export artifacts created in step 2, import generated component code from previous releases as software-in-the-loop (SIL) blocks or processor-in-the-loop (PIL) blocks. For example:

```
crossReleaseImport(P1_R2015bFullPath, 'Controller', ...  
    'SimulationMode', 'SIL');  
crossReleaseImport(P2_R2016aFullPath, 'Controller', ...  
    'SimulationMode', 'SIL');
```


`P1_R2015bFullPath` and `P2_R2016aFullPath` are paths to the export artifact folders, `P1_R2015b` and `P2_R2016a`, created in step 1.

The `crossReleaseImport` function creates software-in-the-loop (SIL) blocks and subfolders:

- `P1_R2015b_sil` and `P1_R2015b_sil_resources`
- `P2_R2016a_sil` and `P2_R2016a_sil_resources`

4 Incorporate components into current release model

Use the Simulink Editor or the `delete_block` and `add_block` commands to replace components with SIL or PIL blocks. For example:

- Replace `P1` with `P1_R2015b_sil`.
- Replace `P2` with `P1_R2016a_sil`.

When you run a model simulation, the simulation runs the previous release code through the SIL or PIL blocks.

When you build the `Controller` model (`rtwbuild('Controller')`), the code generator does not generate new code for the components represented by the SIL or PIL blocks. The model code calls code generated by previous releases.

Limitations

The cross-release code integration workflow does not support:

- Export-function models.
- Model blocks inside the exported component. For a workaround, see “Incorporate Model Reference Code” on page 19-102.
- The model reference code interface for the exported component. The component must use the standalone code interface.
- AUTOSAR code generation.
- Simulink Function, Function Caller, and Data Store Memory blocks across the boundaries of code generated by different releases.
- Parameter tuning through the SIL or PIL blocks.
- The integration of generated code from releases before R2010a.

- The import of generated code from the current release into a previous release (forward compatibility).

At the end of the model build process, the code generation report displays shared files that are directly used by the integration model, for example, `Controller`. The report does not display shared files used by the components of the model, for example, `P1` and `P2`.

Incorporate Model Reference Code

Suppose a Model block in a component references a model through the model reference code interface. When you use `crossReleaseExport` to export generated code from a previous release, the function does not export the referenced model code. You see a warning:

```
Cross-release export does not support model references. Generated code
for the following models will not be exported:
```

To work around this limitation, before running `crossReleaseImport`, use the `sharedCodeUpdate` function to import the generated code for the referenced model into your specified shared code folder.

```
sharedCodeUpdate(referencedModelCodeFolder,existingSharedCodeFolder);
```

The function creates a subfolder for the copied referenced model code.

If you try to import the component code without the referenced model code, `crossReleaseImport` produces a build error.

Simulink.Bus Support

To use a bus object as a data type in cross-release code integration, use one of these approaches.

Approach	Details
Exported bus	<p>In the previous release, before generating code, specify these properties of the <code>Simulink.Bus</code> object:</p> <ul style="list-style-type: none"> • <code>DataScope</code> — Set to <code>Exported</code>. • <code>HeaderFile</code> — Specify a file name, for example, <code>prevRelBusType</code>.

Approach	Details
	<p>The code generator creates <code>prevRelBusType.h</code> in the shared utility code folder. This header file contains the definition for the <code>Simulink.Bus</code> data type. Use <code>sharedCodeUpdate</code> to add <code>prevRelBusType.h</code> to the shared code folder that <code>ExistingSharedCode</code> specifies.</p> <p>For R2010a and R2010b, the <code>DataScope</code> property is not available. Do not assign a value to the <code>HeaderFile</code> property. The code generator creates the <code>Simulink.Bus</code> data type definition in <code>modelName_types.h</code>, which is located in the code generation folder for the model.</p> <p>In the current release, before running <code>crossReleaseImport</code>, set the <code>DataScope</code> property of the <code>Simulink.Bus</code> object to <code>Imported</code>.</p> <p>When you build the integration model that incorporates the imported SIL or PIL block, the build process uses the <code>Simulink.Bus</code> data type definition in <code>prevRelBusType.h</code>.</p> <p>If the imported code is from R2010a or R2010b, specify these properties of the <code>Simulink.Bus</code> object:</p> <ul style="list-style-type: none"> • <code>DataScope</code> — Set to <code>Imported</code>. • <code>HeaderFile</code> — Set to file path for <code>modelName_types.h</code>, which is in the imported code folder. <p>When you build the integration model, the build process uses the <code>Simulink.Bus</code> data type definition in <code>modelName_types.h</code>.</p>
Imported bus	<p>In the previous release, before generating code, specify these properties of the <code>Simulink.Bus</code> object:</p> <ul style="list-style-type: none"> • <code>DataScope</code> — Set to <code>Imported</code>. • <code>HeaderFile</code> — Specify a path to a file that contains the <code>Simulink.Bus</code> data type definition, for example, <code>aBusType.h</code>. <p>For R2010a and R2010b, the <code>DataScope</code> property is not available. For the <code>HeaderFile</code> property, specify a path to a file that contains the <code>Simulink.Bus</code> data type definition, for example, <code>aBusType.h</code>.</p>

Approach	Details
	<p>In the current release, after importing generated code, you do not need to change the <code>Simulink.Bus</code>.</p> <p>When you build the integration model that incorporates the imported SIL or PIL block, the build process uses the <code>Simulink.Bus</code> data type definition from <code>aBusType.h</code>.</p> <p>If the imported code is from R2010a or R2010b, specify these properties of the <code>Simulink.Bus</code> object:</p> <ul style="list-style-type: none"> • <code>DataScope</code> — Set to <code>Imported</code>. • <code>HeaderFile</code> — Set to <code>aBusType.h</code>. <p>When you build the integration model, the build process uses the <code>Simulink.Bus</code> data type definition in <code>aBusType.h</code>.</p>

Parameter Tuning

For a component from a previous release, you can export a block parameter `P` as an inline value in generated code. Before generating code:

- For R2015a and earlier releases, set `InlineParameters` or `InlineParams` to `on`.
- For R2015b and later releases, set `DefaultParameterBehavior` to `Inlined`.

To control tunability in the integration model through a single instance of `P`:

- 1 Before generating code in the previous release, apply the `ExportedGlobal` storage class to the `Simulink.Parameter` object for `P`.
- 2 In the integration model, use the `ImportedExtern` storage class to reference `P`.

You can use the `SimulinkGlobal` storage class for the exported component and integration model. This approach produces separate parameter definitions and values for `P`: one set for the component from the previous release and another set for the rest of the integration model.

See Also

`crossReleaseImport` | `crossReleaseExport` | `sharedCodeUpdate`

Related Examples

- “Cross-Release Shared Code Reuse”

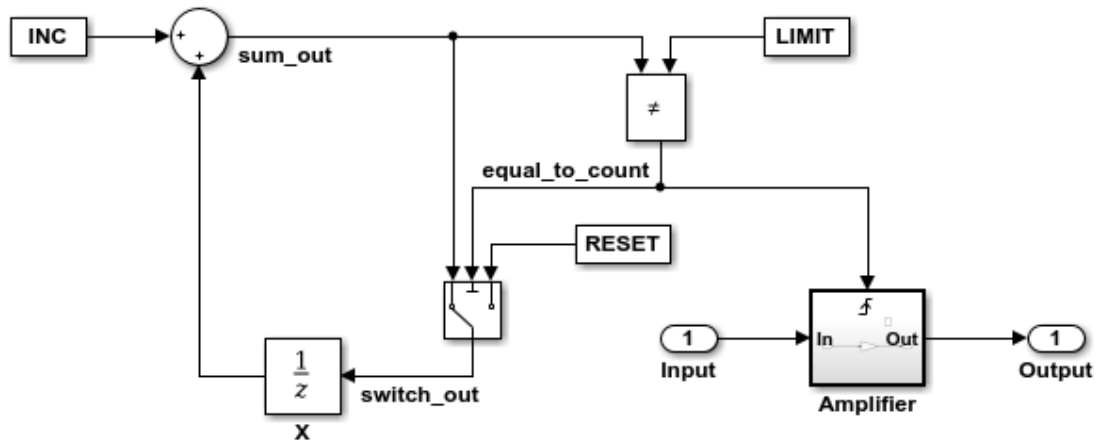
Generate Code Using Simulink® Coder™

This example shows how to select a target for a Simulink® model, generate C code for real-time simulation, and view generated files.

The model represents an 8-bit counter that feeds a triggered subsystem that is parameterized by constant blocks INC, LIMIT, and RESET. Input and Output represent I/O for the model. The Amplifier subsystem amplifies the input signal by gain factor K, which updates when signal equal_to_count is true.

1. Open the model. For example, type the following commands at the MATLAB® command prompt.

```
model='rtwdemo_rtwinintro';
open_system(model)
```



Algorithm Description

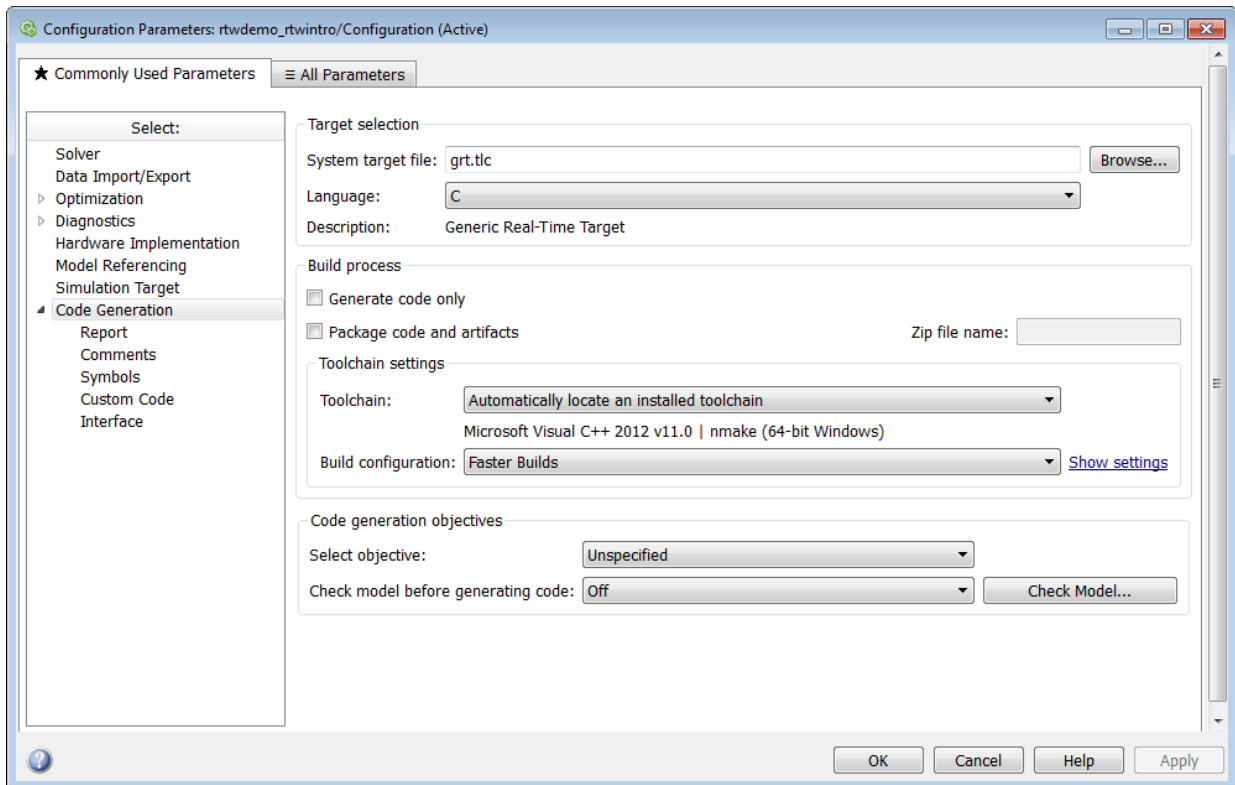
An 8-bit counter feeds a triggered subsystem parameterized by constants INC, LIMIT, and RESET. The I/O for the model is Input and Output. The Amplifier subsystem amplifies the input signal by gain factor K, which is updated whenever signal equal_to_count is true.

2. Open the Configuration Parameters dialog box from the model editor by clicking **Simulation > Configuration Parameters**.

Alternately, type the following commands at the MATLAB® command prompt.

```
cs = getActiveConfigSet(model);
openDialog(cs);
```

3. Select the **Code Generation** node.



4. In the **Target Selection** pane, click **Browse** to select a target.

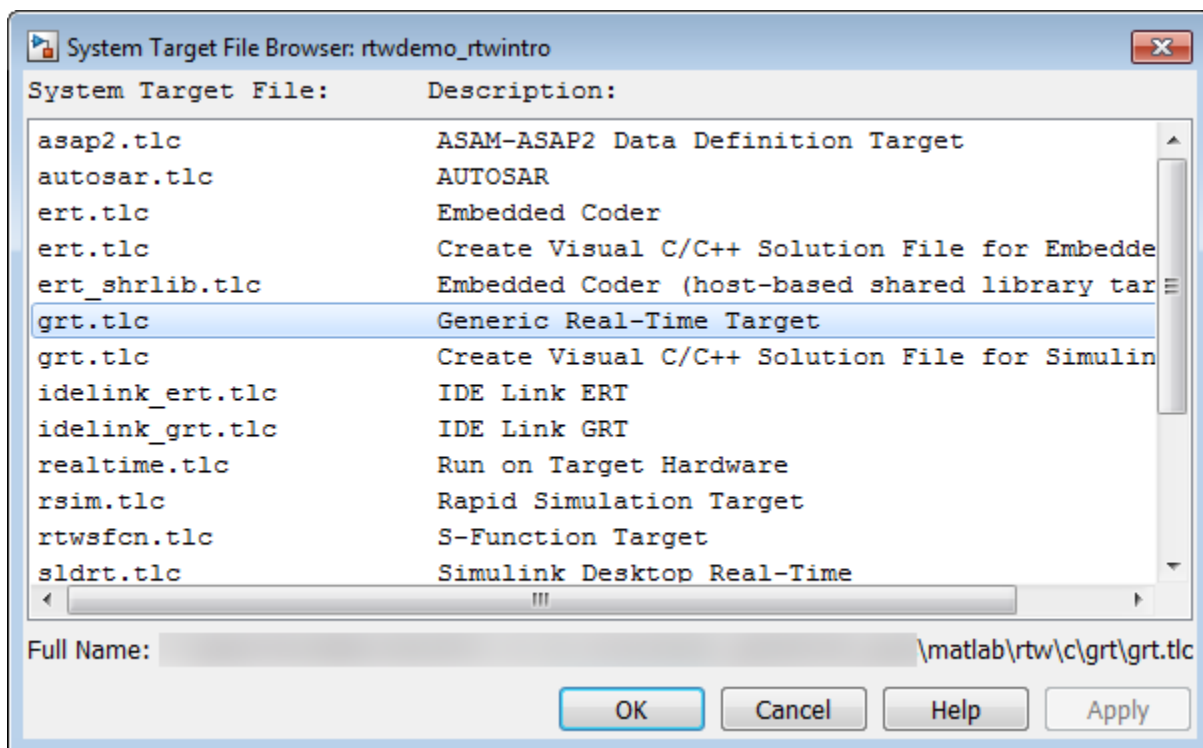
You can generate code for a particular target environment or purpose. Some built-in targeting options are provided using system target files, which control the code generation process for a target.

Target selection

System target file:

Language:

Description: Generic Real-Time Target



5. Select the **Generic Real-Time (GRT)** target and click **Apply**.

Optionally, in the **Code Generation Advisor** pane set the **Select objective** field to **Execution efficiency** or **Debugging**. Then click **Check model...** to identify and systematically change parameters to meet your objectives.

6. In the model window, press **Ctrl+B** to generate code.

Code generation objectives

Select objective:

Check model before generating code:

7. View the code generation report that appears.

The report includes links to model files such as `rtwdemo_rtwintr0.c` and associated utility and header files.

The screenshot shows a window titled "Code Generation Report" for the model "rtwdemo_rtwintr0". The window is divided into a left sidebar and a main content area.

Left Sidebar:

- Contents:**
 - Summary (highlighted)
 - Subsystem Report
 - Code Interface Report
- Generated Code:**
 - [-] Model files
 - [rtwdemo_rtwintr0.c](#)
 - [rtwdemo_rtwintr0.h](#)
 - [rtwdemo_rtwintr0_private.h](#)
 - [rtwdemo_rtwintr0_types.h](#)
 - [+] Shared files (2)
 - [+] Interface files (1)

Main Content Area:

Code Generation Report for 'rtwdemo_rtwintr0'

Summary

Code generation for model "rtwdemo_rtwintr0"

Model version	1.256
Simulink Coder version	8.10 (R2016a) 30-Nov-2015
C source code generated on	Thu Dec 24 10:55:50 2015
C source code generated at	C:\Work\rtwdemo_rtwintr0_grt_rtw\

Configuration settings at the time of code generation: [click to open](#)
 Code generation objectives: **Unspecified**
 Validation result: Not run

At the bottom right of the window are "OK" and "Help" buttons.

The figure below contains a portion of `rtwdemo_rtwintro.c`

```

Step function for model: rtwdemo_rtwintr
File: rtwdemo_rtwintr.c

1  /* Model step function */
2  void rtwdemo_rtwintr_step(void)
3  {
4      uint8_T rtb_sum_out;
5      boolean_T rtb_equal_to_count;
6
7      /* Sum: '<Root>/Sum' incorporates:
8       * Constant: '<Root>/INC'
9       * UnitDelay: '<Root>/X'
10     */
11     rtb_sum_out = (uint8_T)(1U + (uint32_T)rtwdemo_rtwintr_DWork.X);
12
13     /* RelationalOperator: '<Root>/RelOpt' incorporates:
14      * Constant: '<Root>/LIMIT'
15     */
16     rtb_equal_to_count = (rtb_sum_out != 16);
17
18     /* Outputs for Triggered SubSystem: '<Root>/Amplifier' incorporates:
19      * TriggerPort: '<S1>/Trigger'
20     */
21     if (rtb_equal_to_count && (rtwdemo_rtwintr_PrevZCSigState.Amplifier_Trig_ZCE
22      != POS_ZCSIG)) {
23         /* Output: '<Root>/Output' incorporates:
24          * Gain: '<S1>/Gain'
25          * Inport: '<Root>/Input'
26         */
27         rtwdemo_rtwintr_Y.Output = rtwdemo_rtwintr_U.Input << 1;
28     }
29
30     rtwdemo_rtwintr_PrevZCSigState.Amplifier_Trig_ZCE = (uint8_T)
31         (rtb_equal_to_count ? (int32_T)POS_ZCSIG : (int32_T)ZERO_ZCSIG);
32
33     /* End of Outputs for SubSystem: '<Root>/Amplifier' */
34
35     /* Switch: '<Root>/Switch' */
36     if (rtb_equal_to_count) {
37         /* Update for UnitDelay: '<Root>/X' */
38         rtwdemo_rtwintr_DWork.X = rtb_sum_out;
39     } else {
40         /* Update for UnitDelay: '<Root>/X' incorporates:
41          * Constant: '<Root>/RESET'
42         */
43         rtwdemo_rtwintr_DWork.X = 0U;
44     }
45
46     /* End of Switch: '<Root>/Switch' */
47 }

```

8. Close the model.

```
bdclose(model)
rtwdemoclean;
```

Report Generation in Simulink Coder

- “Reports for Code Generation” on page 20-2
- “Generate a Code Generation Report” on page 20-5
- “Generate Code Generation Report After Build Process” on page 20-6
- “Open Code Generation Report” on page 20-8
- “Generate Code Generation Report Programmatically” on page 20-10
- “View Code Generation Report in Model Explorer” on page 20-11
- “Package and Share the Code Generation Report” on page 20-13
- “Document Generated Code with Simulink Report Generator” on page 20-15

Reports for Code Generation

In this section...

“HTML Code Generation Report Location” on page 20-2

“HTML Code Generation Report for Referenced Models” on page 20-3

“HTML Code Generation Report Extensions” on page 20-3

The code generator software produces an HTML code generation report so that you can view and analyze the generated code. When your model is built, the code generation process produces an HTML file that is displayed in an HTML browser or in the Model Explorer. The code generation report includes:

- The **Summary** section lists version, date, and code generation objectives information. The **Configuration settings at the time of code generation** link opens a noneditable view of the Configuration Parameters dialog box. The dialog box shows the Simulink model settings at the time of code generation, including TLC options.
- The **Subsystem Report** section contains information on nonvirtual subsystems in the model.
- In the **Generated Files** section on the **Contents** pane, you can click the names of source code files generated from your model to view their contents in a MATLAB Web browser window. In the displayed source code, global variables are hypertext that links to their definitions.

For an example, see “Generate a Code Generation Report” on page 20-5.

If you have a Simulink Report Generator license, you can document your code generation project in multiple formats, including HTML, PDF, RTF, Microsoft Word, and XML. For an example of how to create a Microsoft Word report, see “Document Generated Code with Simulink Report Generator” on page 20-15.

HTML Code Generation Report Location

The default location for the code generation report files is in the `html` subfolder of the build folder, `model_target_rtw/html/`. `target` is the name of the **System target file** specified on the **Code Generation** pane. The default name for the top-level HTML report file is `model_codegen_rpt.html` or `subsystem_codegen_rpt.html`. For more information on the location of the build folder, see “Control the Location for Generated Files” on page 10-15.

HTML Code Generation Report for Referenced Models

To generate a code generation report for a top model and code generation reports for each referenced model, you need to specify the **Create code generation report** on the **Code Generation > Report** pane for the top model and each referenced model. You can open the code generation report of a referenced model in one of two ways:

- From the top-model code generation report, you can access the referenced model code generation report by clicking a link under **Referenced Models** in the left navigation pane. Clicking a link opens the code generation report for the referenced model in the browser. To navigate back to the top model code generation report, use the **Back** button at the top of the left navigation pane.
- From the referenced model diagram window, select **Code > C/C++ Code > Code Generation Report > Open Model Report**.

For more information, see “Generate Code for Referenced Models” on page 8-4

HTML Code Generation Report Extensions

If you have an Embedded Coder license, the code generator enhances the HTML code generation report. Configure your model to include the following sections in the report:

- The **Code Interface Report** section provides information about the generated code interface, including model entry point functions and input/output data. For more information, see “Analyze the Generated Code Interface”.
- The **Traceability Report** section allows you to account for **Eliminated / Virtual Blocks** that are untraceable versus the listed **Traceable Simulink Blocks / Stateflow Objects / MATLAB Scripts**. This provides a complete mapping between model elements and code. For more information, see “Customize Traceability Reports”.
- The **Static Code Metrics Report** section provides statistics of the generated code. Metrics are estimated from static analysis of the generated code. For more information, see “Static Code Metrics”.
- The **Code Replacements Report** section allows you to account for code replacement library (CRL) functions that were used during code generation, providing a mapping between each replacement instance and the Simulink block that triggered the replacement. For more information, see “Analyze Code Replacements in the Generated Code”.

- The model Web view displays an interactive model diagram within the code generation report and supports traceability between the source code and the model. Therefore, you can share your model and generated code outside of the MATLAB environment. For more information, see “Web View of Model in Code Generation Report”.

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

- If you enable code-to-model traceability, hyperlinks within the displayed source code navigate to the blocks or subsystems from which the code is generated. For more information, see “Traceability in Code Generation Report” and “Trace Code to Model Objects by Using Hyperlinks”.
- If you enable model-to-code traceability, you can navigate to the generated code for a block in the model. For more information, see “Trace Model Objects to Generated Code”.
- If you set the **Code coverage tool** parameter on the **Code Generation > Verification** pane, you can view the code coverage data and annotations. For more information, see “Configure Code Coverage with Third-Party Tools”.
- If you select the **Static code metrics** check box on the **Code Generation > Report** pane, you can view code metrics information and navigate to code definitions and declarations in the generated code. For more information, see “View Static Code Metrics and Definitions Within the Generated Code”.

Related Examples

- “Traceability in Code Generation Report”

Generate a Code Generation Report

To generate a code generation report when the model is built:

- 1** In the Simulink Editor, select **Code > C/C++ Code > Code Generation Report > Options**. The Configuration Parameters dialog box opens with the **Code Generation > Report** pane visible.
- 2** Select the **Create code generation report** parameter.
- 3** If you want the code generation report to automatically open after generating code, select the **Open report automatically** parameter (which is enabled by selecting **Create code generation report**).
- 4** Generate code.

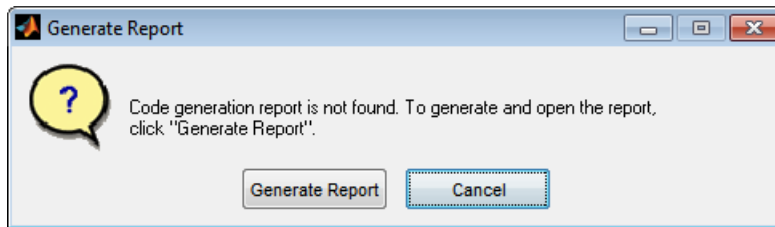
The build process writes the code generation report files to the `html` subfolder of the build folder (see “HTML Code Generation Report Location” on page 20-2). Next, the build process automatically opens a MATLAB Web browser window and displays the code generation report.

To open an HTML code generation report at any time after a build, see “Open Code Generation Report” on page 20-8 and “Generate Code Generation Report After Build Process” on page 20-6.

Generate Code Generation Report After Build Process

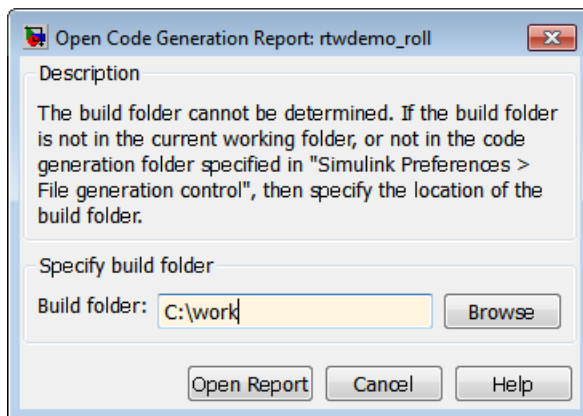
After generating code, if you did not configure your model to create a code generation report, you can generate a code generation report without rebuilding your model.

- 1 In the model diagram window, select **Code > C/C++ Code > Code Generation Report > Open Model Report**.
- 2 If your current working folder contains the code generation files the following dialog opens.



Click **Generate Report**.

- 3 If the code generation files are not in your current working directory, the following dialog opens.



Enter the full path of the build folder for your model, `../model_target_rtw` and click **Open Report**.

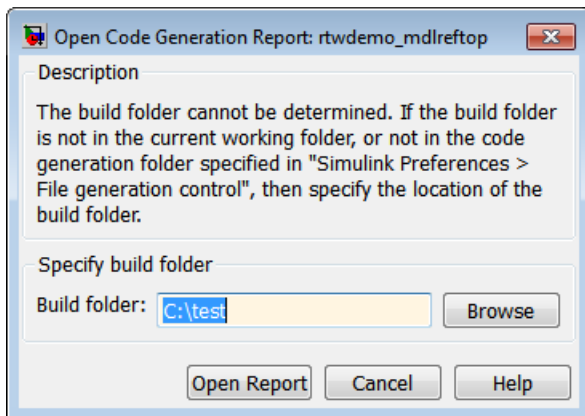
The software generates a report, *model_codgen_rpt.html*, from the code generation files in the build folder you specified.

Note: An alternative method for generating the report after the build process is complete is to configure your model to generate a report and build your model. In this case, the software generates the report without regenerating the code.

Open Code Generation Report

You can refer to existing code generation reports at any time. If you generated a code generation report, in the Simulink Editor, you can open the report by selecting the menu option **Code > C/C++ Code > Code Generation Report > Open Model Report**. If you are opening a report for a subsystem, select **Open Subsystem Report**. A Simulink Coder license is required to view the code generation report. An Embedded Coder license is required to view a code generation report enhanced with Embedded Coder features.

If your current working folder does not contain the code generation files and the code generation report, the following dialog box opens:



Enter the full path of the build folder for your model, `../model_target_rtw` and click **Open Report**.

Alternatively, you can open the code generation report (`model_codegen_rpt.html` or `subsystem_codegen_rpt.html`) manually into a MATLAB Web browser window, or in another Web browser. For the location of the generated report files, see “HTML Code Generation Report Location” on page 20-2.

Limitation

After building your model or generating the code generation report, if you modify legacy or custom code, you must rebuild your model or regenerate the report for the code generation report to include the updated legacy source files. For example, if you modify your legacy code, and then use the **Code > C/C++ Code > Code Generation**

Report > Open Model Report menu to open an existing report, the software does not check if the legacy source file is out of date compared to the generated code. Therefore, the code generation report is not regenerated and the report includes the out-of-date legacy code. This issue also occurs if you open a code generation report using the `coder.report.open` function.

To regenerate the code generation report, do one of the following:

- Rebuild your model.
- Generate the report using the `coder.report.generate` function.

Generate Code Generation Report Programmatically

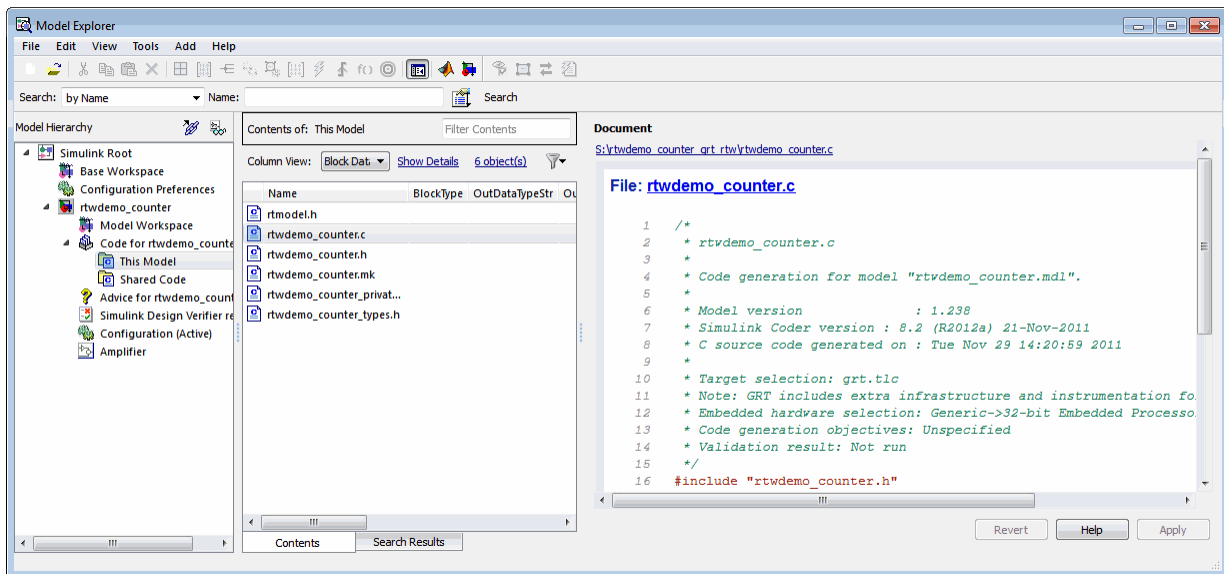
At the MATLAB command line, you can generate, open, and close an HTML Code Generation Report with the following functions:

- `coder.report.generate` generates the code generation report for the specified model.
- `coder.report.open` opens an existing code generation report.
- `coder.report.close` closes the code generation report.

View Code Generation Report in Model Explorer

After generating an HTML code generation report, you can view the report in the right pane of the Model Explorer. You can also browse the generated files directly in the Model Explorer.

When you generate code, or open a model that has generated code for its current target configuration in your working folder, the **Hierarchy** (left) pane of Model Explorer contains a node named **Code for *model***. Under that node are other nodes, typically called **This Model** and **Shared Code**. Clicking **This Model** displays in the **Contents** (middle) pane a list of generated source code files in the build folder of that model. The next figure shows code for the `rtwdemo_counter` model.



In this example, the file `S:/rtwdemo_counter_grt_rtw/rtwdemo_counter.c` is being displayed. To view a file in the **Contents** pane, click it once.

The views in the **Document** (right) pane are read only. The code listings there contain hyperlinks to functions and macros in the generated code. Clicking the file hyperlink opens that source file in a text editing window where you can modify its contents.

If an open model contains Model blocks, and if generated code for these models exists in the current `slprj` folder, nodes for the referenced models appear in the **Hierarchy** pane

one level below the node for the top model. Such referenced models do not need to be open for you to browse and read their generated source files.

If the code generator produces shared utility code for a model, a node named **Shared Code** appears directly under the **This Model** node. It collects source files that exist in the `./slprj/target/_sharedutils` subfolder.

Note You cannot use the **Search** tool built into Model Explorer toolbar to search generated code displayed in the Code Viewer. On PCs, typing **Ctrl+F** when focused on the **Document** pane opens a Find dialog box that you can use to search for text in the currently displayed file. You can also search for text in the HTML report window, and you can open the files in the editor.

Package and Share the Code Generation Report

In this section...

“Package the Code Generation Report” on page 20-13

“View the Code Generation Report” on page 20-14

Package the Code Generation Report

To share the code generation report, you can package the code generation report files and supporting files into a zip file for transfer. The default location for the code generation report files is in two folders:

- /slprj
- html subfolder of the build folder, *model_target_rtw*, for example *rtwdemo_counter_grt_rtw/html*

To create a zip file from the MATLAB command window:

- 1 In the Current Folder browser, select the two folders:
 - /slprj
 - Build folder: *model_target_rtw*
- 2 Right-click to open the context menu.
- 3 In the context menu, select **Create Zip File**. A file appears in the Current Folder browser.
- 4 Name the zip file.

Alternatively, you can use the MATLAB `zip` command to zip the code generation report files:

```
zip('myzip',{'slprj','rtwdemo_counter_grt_rtw'})
```

Note: If you need to relocate the static and generated code files for a model to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB and Simulink products, use the code generator pack-and-go utility. For more information, see “Relocate Code to Another Development Environment” on page 25-51.

View the Code Generation Report

To view the code generation report after transfer, unzip the file and save the two folders at the same folder level in the hierarchy. Navigate to the *model_target_rtw/html/* folder and open the top-level HTML report file named *model_codgen_rpt.html* or *subsystem_codegen_rpt.html* in a Web browser.

Document Generated Code with Simulink Report Generator

In this section...

- “Generate Code for the Model” on page 20-16
- “Open the Report Generator” on page 20-16
- “Set Report Name, Location, and Format” on page 20-18
- “Include Models and Subsystems in a Report” on page 20-19
- “Customize the Report” on page 20-20
- “Generate the Report” on page 20-21

The Simulink Report Generator software creates documentation from your model in multiple formats, including HTML, PDF, RTF, Microsoft Word, and XML. This example shows one way to document a code generation project in Microsoft Word. The generated report includes:

- System snapshots (model and subsystem diagrams)
- Block execution order list
- Simulink Coder and model version information for generated code
- List of generated files
- Optimization configuration parameter settings
- System target file selection and build process configuration parameter settings
- Subsystem map
- File name, path, and generated code listings for the source code

To adjust Simulink Report Generator settings to include custom code and then generate a report for a model, complete the following tasks:

- 1 “Generate Code for the Model” on page 20-16
- 2 “Open the Report Generator” on page 20-16
- 3 “Set Report Name, Location, and Format” on page 20-18
- 4 “Include Models and Subsystems in a Report” on page 20-19
- 5 “Customize the Report” on page 20-20
- 6 “Generate the Report” on page 20-21

A Simulink Report Generator license is required for the following report formats: PDF, RTF, Microsoft Word, and XML. For more information on generating reports in these formats, see the Simulink Report Generator documentation.

Generate Code for the Model

Before you use the Report Generator to document your project, generate code for the model.

- 1 In the MATLAB Current Folder browser, navigate to a folder where you have write access.
- 2 Create a working folder from the MATLAB command line by typing:

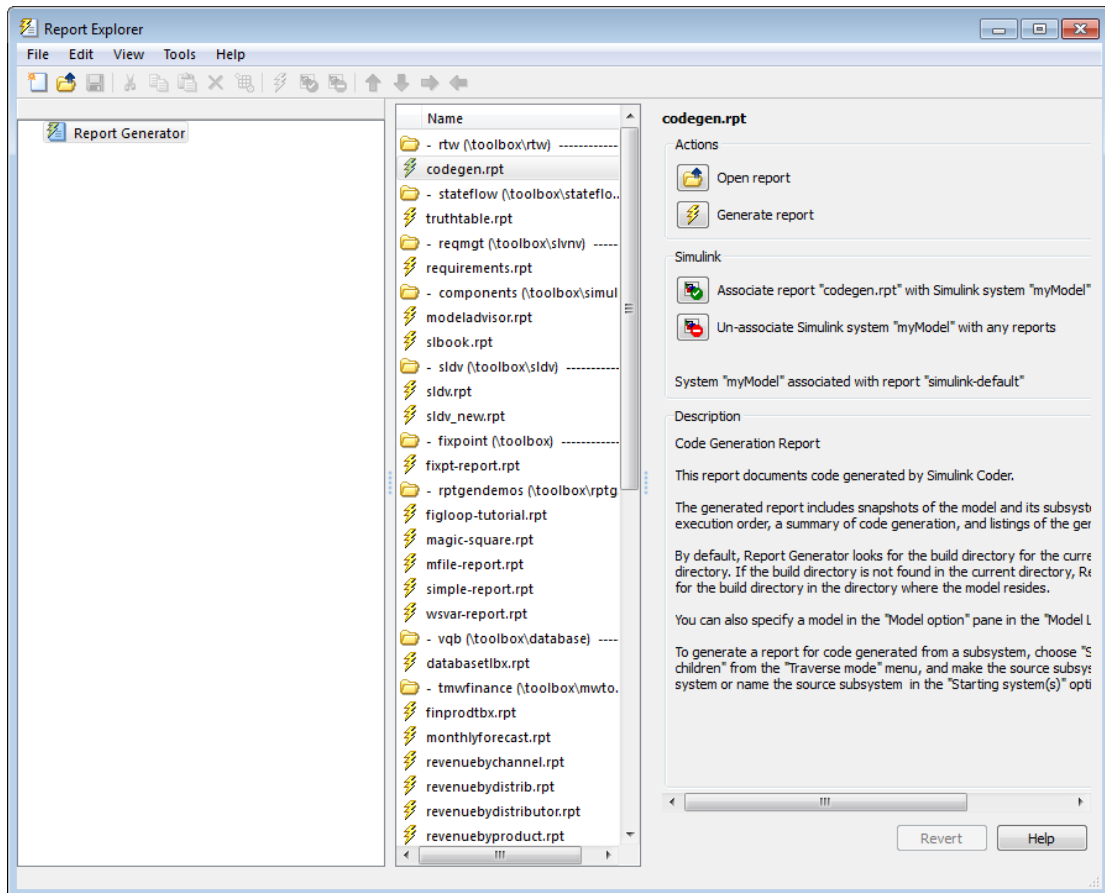
```
mkdir report_ex
```
- 3 Make `report_ex` your working folder:

```
cd report_ex
```
- 4 Open the `slexAircraftExample` model by entering the model name on the MATLAB command line.
- 5 In the model window, choose **File > Save As**, navigate to the working folder, `report_ex`, and save a copy of the `slexAircraftExample` model as `myModel`.
- 6 Open the Configuration Parameters dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu.
- 7 Select the **Solver** pane. In the **Solver options** section, specify the **Type** parameter as **Fixed-step**.
- 8 Select the **Code Generation** pane. Select **Generate code only**.
- 9 Click **Apply**.
- 10 In the model window, press **Ctrl+B**. The build process generates code for the model.


Open the Report Generator

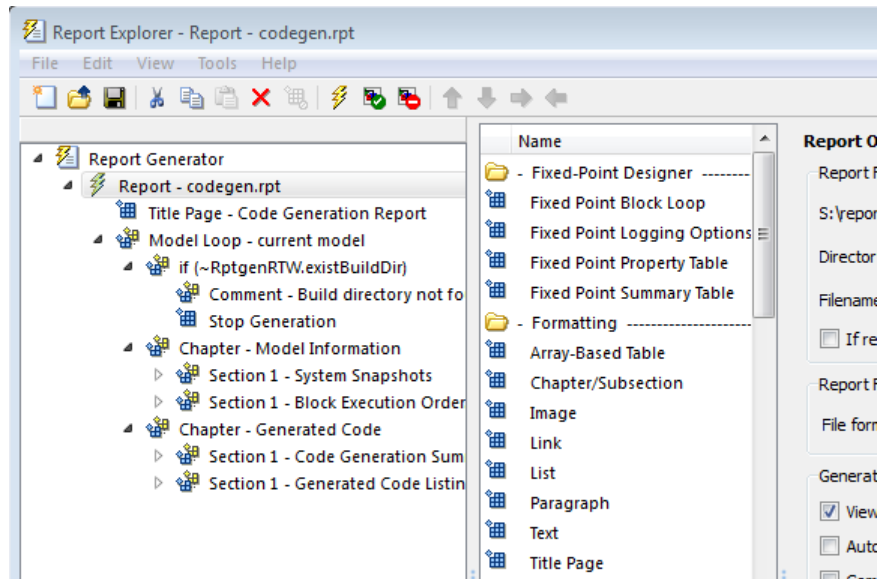
After you generate the code, open the Report Generator.

- 1 In the model diagram window, select **Tools > Report Generator**.
- 2 In the Report Explorer window, in the options pane (center), click the folder `rtw` (`\toolbox\rtw`). Click the setup file that it contains, `codegen.rpt`.



3

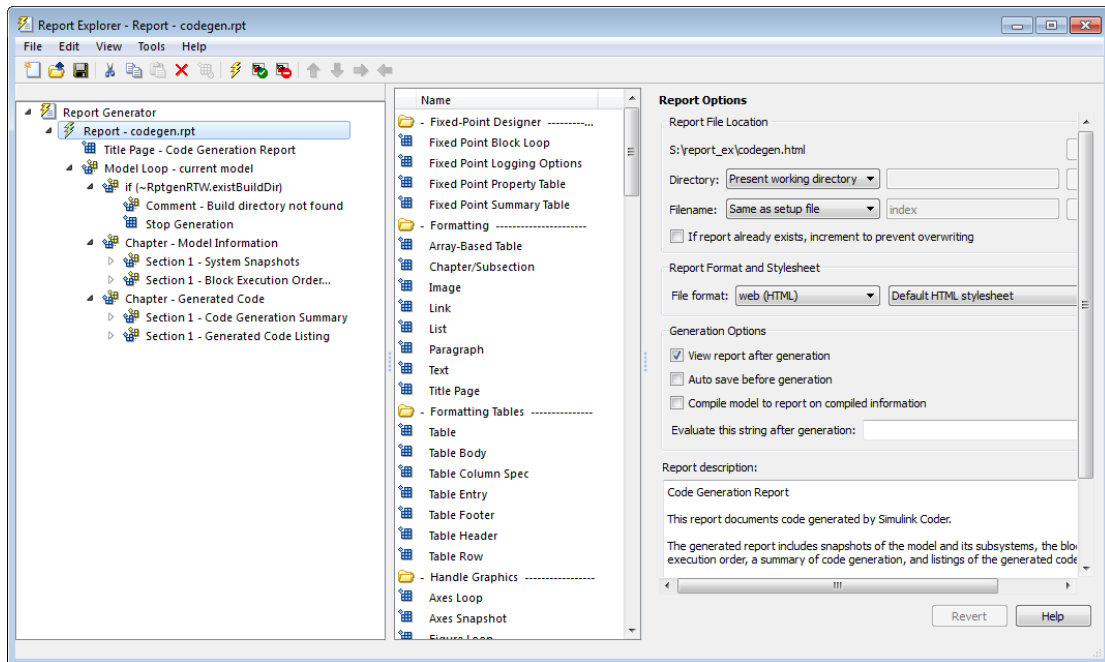
Double-click **codegen.rpt** or select it and click the **Open report** button . The Report Explorer displays the structure of the setup file in the outline pane (left).



Set Report Name, Location, and Format

Before generating a report, you can specify report output options, such as the folder, file name, and format. For example, to generate a Microsoft Word report named `MyCGModelReport.rtf`:

- 1 In the properties pane, under **Report Options**, review the options listed.

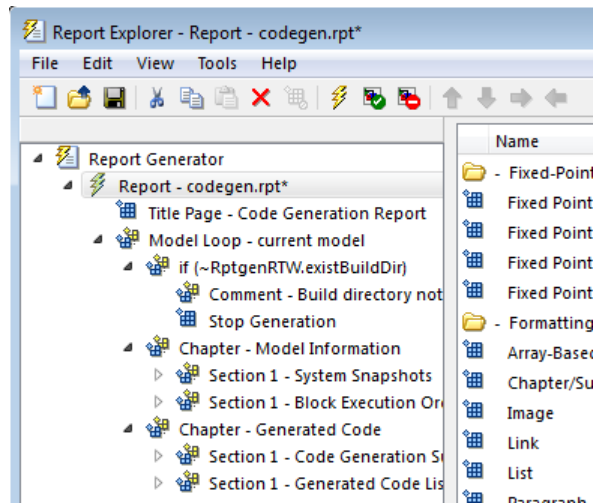


- 2 Leave the **Directory** field set to **Present working directory**.
- 3 For **Filename**, select **Custom:** and replace **index** with the name **MyModelICGreport**.
- 4 For **File format**, specify **Rich Text Format** and replace **Standard Print** with **Numbered Chapters & Sections**.

Include Models and Subsystems in a Report

Specify the models and subsystems that you want to include in the generated report by setting options in the Model Loop component.

- 1 In the outline pane (left), select **Model Loop**. Report Generator displays Model Loop component options in the properties pane.
- 2 If not already selected, select **Current block diagram** for the **Model name** option.
- 3 In the outline pane, click **Report - codegen.rpt***.



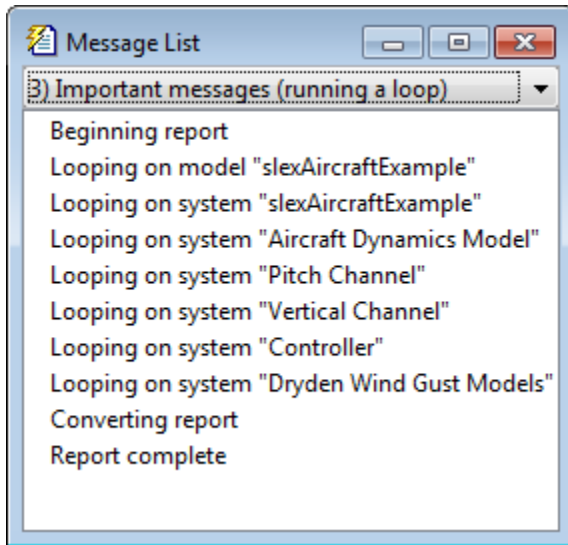
Customize the Report

After specifying the models and subsystems to include in the report, you can customize the sections included in the report.

- 1 In the outline pane (left), expand the node **Chapter - Generated Code**. By default, the report includes two sections, each containing one of two report components.
- 2 Expand the node **Section 1 — Code Generation Summary**.
- 3 Select **Code Generation Summary**. Options for the component are displayed in the properties pane.
- 4 Click **Help** to review the report customizations that you can make with the Code Generation Summary component. For this example, do not customize the component.
- 5 In the Report Explorer window, expand the node **Section 1 — Generated Code Listing**.
- 6 Select **Import Generated Code**. Options for the component are displayed in the properties pane.
- 7 Click **Help** to review the report customizations that you can make with the Import Generated Code component.

Generate the Report

After you adjust the report options, from the **Report Explorer** window, generate the report by clicking **File > Report**. A **Message List** dialog box opens, which displays messages that you can monitor as the report is generated. Model snapshots also appear during report generation. The **Message List** dialog box might be hidden behind other dialog boxes.



When the report is complete, open the report, `MyModelCGReport.rtf` in the folder `report_ex` (in this example).

Code Replacement in Simulink Coder

- “What Is Code Replacement?” on page 21-2
- “Choose a Code Replacement Library” on page 21-8
- “Replace Code Generated from Simulink Models” on page 21-10

What Is Code Replacement?

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
 - Elimination of `math.h`.
 - Elimination of system header files.
 - Elimination of calls to `memcpy` or `memset`.
 - Use of BLAS.
 - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from the following libraries that MathWorks provides:

- GNU C99 extensions—GNU⁶ gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
- AUTOSAR 4.0—Produces code that more closely aligns with the AUTOSAR standard. Requires an Embedded Coder license.
- Intel IPP for x86-64 (Windows)—Generates calls to the Intel[®] Performance Primitives (IPP) library for the x86-64 Windows platform.
- Intel IPP/SSE for x86-64 (Windows)—Generates calls to the IPP and Streaming SIMD Extensions (SSE) libraries for the x86-64 Windows platform.
- Intel IPP for x86-64 (Windows using MinGW compiler)—Generates calls to the IPP library for the x86-64 Windows platform and MinGW compiler.

6. GNU is a registered trademark of the Free Software Foundation.

- Intel IPP/SSE for x86-64 (Windows using MinGW compiler)—Generates calls to the IPP and SSE libraries for the x86-64 Windows platform and MinGW compiler.
- Intel IPP for x86/Pentium (Windows)—Generates calls to the IPP library for the x86/Pentium Windows platform.
- Intel IPP/SSE for x86/Pentium (Windows)—Generates calls to the Intel Performance IPP and SSE libraries for the x86/Pentium Windows platform.
- Intel IPP for x86-64 (Linux)—Generates calls to the IPP library for the x86-64 Linux platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)—Generates calls to the GNU libraries for IPP and SSE, with GNU C99 extensions, for the x86-64 Linux platform.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available. If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

Code Replacement Libraries

A *code replacement library* consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A *code replacement table* contains one or more *code replacement entries*, with each entry representing a potential replacement for a function or operator. Each entry maps a *conceptual representation* of a function or operator to an *implementation representation* and priority.

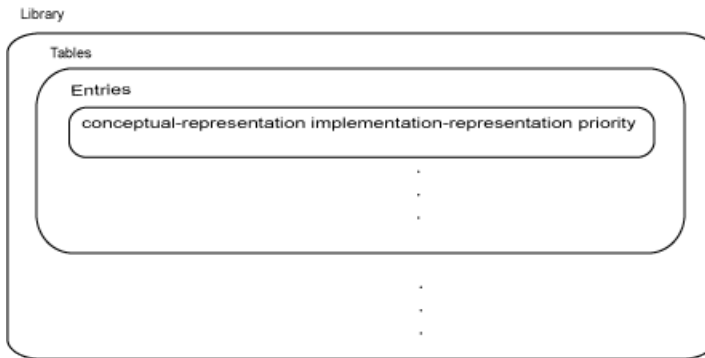


Table Entry Component	Description
Conceptual representation	<p>Identifies the table entry and contains match criteria for the code generator. Consists of:</p> <ul style="list-style-type: none"> • Function name or a key. The function name identifies most functions. For operators and some functions, a series of characters, called a key identifies a function or operator. For example, function name 'COS' and operator key 'RTW_OP_ADD'. • Conceptual arguments that observe code generator naming ('y1', 'u1', 'u2', ...), with corresponding I/O types (output or input) and data types. • Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.
Implementation representation	<p>Specifies replacement code. Consists of:</p> <ul style="list-style-type: none"> • Function name. For example, 'cos_db1' or 'u8_add_u8_u8'. • Implementation arguments, with corresponding I/O types (output or input) and data types. • Parameters that provide additional implementation details, such as header and source file names and paths of build resources.

Table Entry Component	Description
Priority	Defines the entry priority relative to other entries in the table. The value can range from 0 to 100, with 0 being the highest priority. If multiple entries have the same priority, the code generator uses the first match with that priority.

When the code generator looks for a match in a code replacement library, it creates and populates a *call site object* with the function or operator conceptual representation. If a match exists, the code generator uses the matched code replacement entry populated with the implementation representation and uses it to generate code.

The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. If the code generator finds multiple matches within a table, the priority determines the match. The code generator uses a higher-priority entry over a similar entry with a lower priority.

Code Replacement Terminology

Term	Definition
Cache hit	A code replacement entry for a function or operator, defined in the specified code replacement library, for which the code generator finds a match.
Cache miss	A conceptual representation of a function or operator for which the code generator does not find a match.
Call site object	Conceptual representation of a function or operator that the code generator uses when it encounters a call site for a function or operator. The code generator uses the object to query the code replacement library for a conceptual representation match. If a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority, and uses that object to generate replacement code.
Code replacement library	One or more code replacement tables that specify application-specific implementations of functions

Term	Definition
	and operators. When configured to use a code replacement library, the code generator uses criteria defined in the library to search for matches. If a match is found, the code generator replaces code that it generates by default with application-specific code defined in the library.
Code replacement table	One or more code replacement table entries. Provides a way to group related or shared entries for use in different libraries.
Code replacement entry	Represents a potential replacement for a function or operator. Maps a conceptual representation of a function or operator to an implementation representation and priority.
Conceptual argument	Represents an input or output argument for a function or operator being replaced. Conceptual arguments observe naming conventions ('y1', 'u1', 'u2', ...) and data types familiar to the code generator.
Conceptual representation	Represents match criteria that the code generator uses to qualify functions and operators for replacement. Consists of: <ul style="list-style-type: none"> • Function or operator name or key • Conceptual arguments with type, dimension, and complexity specification for inputs and output • Attributes, such as an algorithm and fixed-point saturation and rounding modes
Implementation argument	Represents an input or output argument for a C or C++ replacement function. Implementation arguments observe C/C++ name and data type specifications.

Term	Definition
Implementation representation	<p>Specifies C or C++ replacement function prototype. Consists of:</p> <ul style="list-style-type: none"> • Function name (for example, 'cos_db1' or 'u8_add_u8_u8') • Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output • Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags
Key	<p>Identifies a function or operator that is being replaced. A function name or key appears in the conceptual representation of a code replacement entry. The key RTW_OP_ADD identifies the addition operator.</p>
Priority	<p>Defines the match priority for a code replacement entry relative to other entries, which have the same name and conceptual argument list, within a code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If a library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.</p>

Code Replacement Limitations

Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code.

Related Examples

- “Choose a Code Replacement Library” on page 21-8
- “Replace Code Generated from Simulink Models” on page 21-10

Choose a Code Replacement Library

In this section...

“About Choosing a Code Replacement Library” on page 21-8

“Explore Available Code Replacement Libraries” on page 21-8

“Explore Code Replacement Library Contents” on page 21-8

About Choosing a Code Replacement Library

By default, the code generator does not use a code replacement library.

If you are considering using a code replacement library:

- 1 Explore available libraries. Identify one that best meets your application needs.
 - Consider the lists of application code replacement requirements and libraries that MathWorks provides in “What Is Code Replacement?” on page 21-2.
 - See “Explore Available Code Replacement Libraries” on page 21-8.
- 2 Explore the contents of the library. See “Explore Code Replacement Library Contents” on page 21-8.

If you do not find a suitable library and you have an Embedded Coder license, you can create a custom code replacement library. For more information, see “What Is Code Replacement Customization?” in the Embedded Coder documentation.

Explore Available Code Replacement Libraries

Select the “Code replacement library” to use for code generation from the **Configuration Parameters > Code Generation > Interface** pane. To view a description of a library, select and hover your cursor over the library name. A tooltip describes the library and lists the tables that it contains. The tooltip lists the tables in the order that the code generator searches for a function or operator match.

Explore Code Replacement Library Contents

Use the Code Replacement Viewer to explore the content of a code replacement library.

- 1 At the command prompt, type `crviewer`.

```
>> crviewer
```

The viewer opens. To view the content of a specific library, specify the name of the library as an argument in single quotes. For example:

```
>> crviewer('GNU C99 extensions')
```

- 2 In the left pane, select the name of a library. The viewer displays information about the library in the right pane.
- 3 In the left pane, expand the library, explore the list of tables it contains, and select a table from the list. In the middle pane, the viewer displays the function and operator entries that are in that table, along with abbreviated information for each entry.
- 4 In the middle pane, select a function or operator. The viewer displays information from the table entry in the right pane.

If you select an operator entry that specifies net slope fixed-point parameters (instantiated from entry class `RTW.Tf1COperationEntryGenerator` or `RTW.Tf1COperationEntryGenerator_NetSlope`), the viewer displays an additional tab that shows fixed-point settings.

See Code Replacement Viewer for details on what the viewer displays.

Related Examples

- “What Is Code Replacement?” on page 21-2
- “Replace Code Generated from Simulink Models” on page 21-10

Replace Code Generated from Simulink Models

This example shows how to replace generated code, using a code replacement library. Code replacement is a technique you can use to change the code that the code generator produces for functions and operators to meet application code requirements.

Prepare for Code Replacement

- 1 Make sure that MATLAB, Simulink, Simulink Coder, and a C compiler are installed on your system. Some code replacement libraries available in your development environment can also require Embedded Coder.

To install MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to see which other MathWorks products are installed, in the Command Window, enter `ver`.

- 2 Identify an existing or create a Simulink model for which you want the code generator to replace code.

Choose a Code Replacement Library

If you are not sure which library to use, explore the available libraries.

Configure Code Generator To Use Code Replacement Library

- 1 Configure the code generator to apply a code replacement library during code generation for the model. Do one of the following:
 - In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, select a library for the “Code replacement library” parameter.
 - Set the `CodeReplacementLibrary` parameter at the command line or programmatically.
- 2 Configure the code generator to produce code only (not build an executable) so you can verify your code replacements before building an executable. Do one of the following:
 - In the Configuration Parameters dialog box, on the **Code Generation** pane, select “Generate code only”.
 - Set the `GenCodeOnly` parameter at the command line or programmatically.

Include Code Replacement Information In Code Generation Report

If you have an Embedded Coder license, you can configure the code generator to include a code replacement section in the code generation report. The additional information can help you verify code replacements.

- 1 Configure the code generator to generate a report. In the Configuration Parameters dialog box, on the **Code Generation > Report** pane, select “Create code generation report”. Consider having the report open automatically. Select “Open report automatically”.
- 2 Include the code replacement section in the report. On the **All Parameters** tab, select “Summarize which blocks triggered code replacements”.

Generate Replacement Code

Generate C/C++ code from the model and, if you configured the code generator accordingly, a code generation report. For example, in the model window, press **Ctrl+B**.

The code generator produces the code and displays the report.

Verify Code Replacements

Verify code replacements by examining the generated code. It is possible that code replacement behaves differently than you expect. For example, data types that you observe in the code generator input might not match what the code generator uses as intermediate data types during an operation.

Related Examples

- “What Is Code Replacement?” on page 21-2
- “Choose a Code Replacement Library” on page 21-8
- “Code Generation Configuration” on page 16-2

Deployment

Desktops in Simulink Coder

- “Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” on page 22-2
- “Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target” on page 22-33

Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File

After you create a model, you can use the rapid simulation (RSim) system target file to characterize model behavior. The executable program that results from the build process is for non-real-time execution on your development computer. The executable program is highly optimized for simulating models of hybrid dynamic systems, including models that use variable-step solvers and zero-crossing detection. The speed of the generated code makes the RSim system target file ideal for building programs for batch or Monte Carlo simulations.

About Rapid Simulation

Use the RSim target to generate an executable that runs fast, standalone simulations. You can repeat simulations with varying data sets, interactively or programmatically with scripts, without rebuilding the model. This can accelerate the characterization and tuning of model behavior and code generation testing.

Using command-line options:

- Define parameter values and input signals in one or more MAT-files that you can load and reload at the start of simulations without rebuilding your model.
- Redirect logging data to one or more MAT-files that you can then analyze and compare.
- Control simulation time.
- Specify external mode options.

Note: To run an RSim executable, configure your computer to run MATLAB and have the MATLAB and Simulink installation folders accessible. To deploy a standalone host executable (i.e., without MATLAB and Simulink installed), consider using the Host-Based Shared Library target (`ert_shrlib`)."

Rapid Simulation Advantage

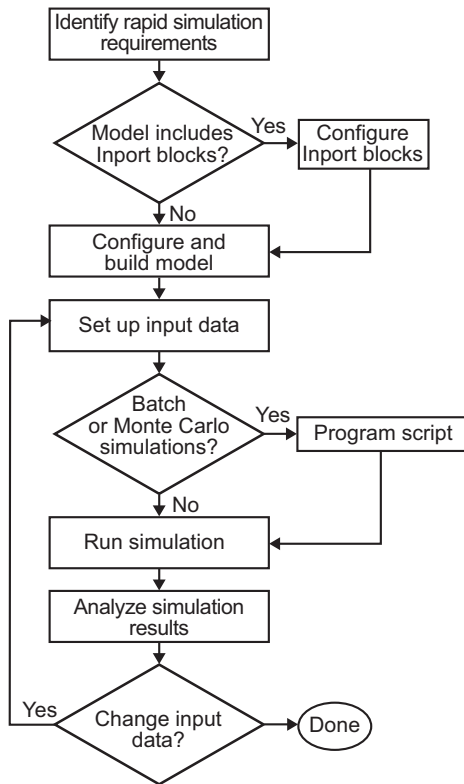
The advantage that you gain from rapid simulation varies. Larger simulations achieve speed improvements of up to 10 times faster than standard Simulink simulations. Some

models might not show noticeable improvement in simulation speed. To determine the speed difference for your model, time your standard Simulink simulation and compare the results with a rapid simulation. In addition, test the model simulation in Rapid Accelerator simulation mode.

General Rapid Simulation Workflow

Like other stages of Model-Based Design, characterization and tuning of model behavior is an iterative process, as shown in the general workflow diagram in the figure. Tasks in the workflow are:

- 1 Identify your rapid simulation requirements.
- 2 Configure Inport blocks that provide input source data for rapid simulations.
- 3 Configure the model for rapid simulation.
- 4 Set up simulation input data.
- 5 Run the rapid simulations.



Identify Rapid Simulation Requirements

The first step to setting up a rapid simulation is to identify your simulation requirements.

Question...	For More Information, See...
How long do you want simulations to run?	“Configure and Build Model for Rapid Simulation” on page 22-6
Are there solver requirements? Do you expect to use the same solver for which the model is configured for your rapid simulations?	“Configure and Build Model for Rapid Simulation” on page 22-6

Question...	For More Information, See...
Do your rapid simulations need to accommodate flexible custom code interfacing? Or, do your simulations need to retain storage class settings?	“Configure and Build Model for Rapid Simulation” on page 22-6
Will you be running simulations with multiple data sets?	“Set Up Rapid Simulation Input Data” on page 22-8
Will the input data consist of global parameters, signals, or both?	“Set Up Rapid Simulation Input Data” on page 22-8
What type of source blocks provide input data to the model — From File, Inport, From Workspace?	“Set Up Rapid Simulation Input Data” on page 22-8
Will the model's parameter vector (<i>model_P</i>) be used as input data?	“Create a MAT-File That Includes a Model Parameter Structure” on page 22-9
What is the data type of the input parameters and signals?	“Set Up Rapid Simulation Input Data” on page 22-8
Will the source data consist of one variable or multiple variables?	“Set Up Rapid Simulation Input Data” on page 22-8
Does the input data include tunable parameters?	“Create a MAT-File That Includes a Model Parameter Structure” on page 22-9
Do you need to gain access to tunable parameter information — model checksum and parameter data types, identifiers, and complexity?	“Create a MAT-File That Includes a Model Parameter Structure” on page 22-9
Will you have a need to vary the simulation stop time for simulation runs?	“Configure and Build Model for Rapid Simulation” on page 22-6 and “Override a Model Simulation Stop Time” on page 22-21
Do you want to set a time limit for the simulation? Consider setting a time limit if your model experiences frequent zero crossings and has a small minor step size.	“Set a Clock Time Limit for a Rapid Simulation” on page 22-21
Do you need to preserve the output of each simulation run?	“Specify a New Output File Name for a Simulation” on page 22-29 and “Specify New Output File Names for To File Blocks” on page 22-29

Question...	For More Information, See...
Do you expect to run the simulations interactively or in batch mode?	“Scripts for Batch and Monte Carlo Simulations” on page 22-18

Configure Inports to Provide Simulation Source Data

You can use Inport blocks as a source of input data for rapid simulations. To do so, configure the blocks so that they can import data from external MAT-files. By default, the Inport block inherits parameter settings from downstream blocks. In most cases, to import data from an external MAT-file, you must explicitly set the following parameters to match the source data in the MAT-file.

- **Main > Interpolate data**
- **Signal Attributes > Port dimensions**
- **Signal Attributes > Data type**
- **Signal Attributes > Signal type**

If you do not have control over the model content, you might need to modify the data in the MAT-file to conform to what the model expects for input. Input data characteristics and specifications of the Inport block that receives the data must match.

For details on adjusting these parameters and on creating a MAT-file for use with an Inport block, see “Create a MAT-File for an Inport Block” on page 22-14. For descriptions of the preceding block parameters, see the description of the Inport block in the Simulink documentation.

Configure and Build Model for Rapid Simulation

After you identify your rapid simulation requirements, configure the model for rapid simulation.

- 1 Open the Configuration Parameters dialog box.
- 2 Go to the **Code Generation** pane.
- 3 On the **Code Generation** pane, click **Browse**. The System Target File Browser opens.
- 4 Select `rsim.tlc` (Rapid Simulation Target) and click **OK**.

On the **Code Generation** pane, the code generator populates the **Make command** and “Template makefile” fields with default settings and adds the **RSim Target** pane under **Code Generation**.

- Click **RSim Target** to view the **RSim Target** pane.

The screenshot shows the RSim Target configuration pane with the following settings:

- Parameter loading:** Enable RSim executable to load parameters from a MAT-file
- Solver:** Solver selection:
- Storage classes:** Force storage classes to AUTO

- Set the RSim target configuration parameters to your rapid simulation requirements.

If You Want to...	Then...
Generate code that allows the RSim executable to load parameters from a MAT-file	Select Enable RSim executable to load parameters from a MAT-file (default).
Let the target choose a solver based on the solver already configured for the model	Set Solver selection to auto (default). The code generator uses a built-in solver if a fixed-step solver is specified on the Solver pane or calls the Simulink solver module (a shared library) if a variable-step solver is specified.
Explicitly instruct the target to use a fixed-step solver	Set Solver selection to Use fixed-step solvers . In the Configuration Parameters dialog box, on the Solver pane, specify a fixed-step solver.
Explicitly instruct the target to use a variable-step solver	Set Solver selection to Use Simulink solver module . In the Configuration Parameters dialog box, on the Solver pane, specify a variable-step solver.
Force storage classes to Auto for flexible custom code interfacing	Select Force storage classes to AUTO (default).
Retain storage class settings, such as ExportedGlobal or ImportedExtern , due to application requirements	Clear Force storage classes to AUTO .

- 7 Set up data import and export options. On the **Data Import/Export** pane, in the **Save to Workspace** section, select the **Time**, **States**, **Outputs**, and **Final States** options, as they apply. By default, the code generator saves simulation logging results to a file named *model.mat*. For more information, see “Export Simulation Data”.
- 8 If you are using external mode communication, set up the interface, using the **Code Generation > Interface** pane. See “What You Can Do With a Host/Target Communication Channel” on page 26-2 for details.
- 9 Press **Ctrl+B**. The code generator builds a highly optimized executable program that you can run on your development computer with varying data, without rebuilding.

For more information on compilers that are compatible with the Simulink Coder product, see “Select and Configure Compiler or IDE” on page 25-3 and “Template Makefiles and Make Options” on page 17-26 .

Set Up Rapid Simulation Input Data

- “About Rapid Simulation Data Setup” on page 22-8
- “Create a MAT-File That Includes a Model Parameter Structure” on page 22-9
- “Create a MAT-File for a From File Block” on page 22-13
- “Create a MAT-File for an Inport Block” on page 22-14

About Rapid Simulation Data Setup

The format and setup of input data for a rapid simulation depends on your requirements.

If the Input Data Source Is...	Then...
The model's global parameter vector (<i>model_P</i>)	Use the <code>rsimgetrtp</code> function to get the vector content and then save it to a MAT-file.
The model's global parameter vector and you want a mapping between the vector and tunable parameters	Call the <code>rsimgetrtp</code> function to get the global parameter structure and then save it to a MAT-file.
Provided by a From File block	Create a MAT-file that a From File block can read.
Provided by an Inport block	Create a MAT-file that adheres to one of the three data file formats that the Inport block can read.

If the Input Data Source Is...	Then...
Provided by a From Workspace block	Create structure variables in the MATLAB workspace.

The RSim target requires that MAT-files used as input for From File and Inport blocks contain data. The `grt` target inserts MAT-file data directly into the generated code, which is then compiled and linked as an executable. In contrast, RSim allows you to replace data sets for each successive simulation. A MAT-file containing From File or Inport block data must be present if a From File block or Inport block exists in your model.

Create a MAT-File That Includes a Model Parameter Structure

To create a MAT-file that includes a model global parameter structure (*model_P*),

- 1 Get the structure by calling the function `rsimgetrtp`.
- 2 Save the parameter structure to a MAT-file.

If you want to run simulations over varying data sets, consider converting the parameter structure to a cell array and saving the parameter variations to a single MAT-file.

Get the Parameter Structure for a Model

Get the global parameter structure (*model_P*) for a model by calling the function `rsimgetrtp`.

```
param_struct = rsimgetrtp('model')
```

Argument	Description
<i>model</i>	The model for which you are running the rapid simulations.

The `rsimgetrtp` function forces an update diagram action for the specified model and returns a structure that contains the following fields.

Field	Description
<code>modelChecksum</code>	A four-element vector that encodes the structure of the model. The code generator uses the checksum to check whether the structure of the model has changed since the RSim executable was generated. If you delete or add a block, and then generate a new <i>model_P</i> vector, the new checksum does not match the

Field	Description
	original checksum anymore. The RSim executable detects this incompatibility in parameter vectors and exits to avoid returning incorrect simulation results. If the model structure changes, you must regenerate the code for the model.
parameters	A structure that contains the model's global parameters.

The parameter structure contains the following information.

Field	Description
dataTypeName	The name of the parameter data type, for example, <code>double</code>
dataTypeID	Internal data type identifier used by the code generator
complex	The value 0 if real; 1 if complex
dtTransIdx	Internal data index used by the code generator
values	A vector of the parameter values associated with this structure
map	This field contains the mapping information that correlates the 'values' to the tunable parameters of the model. This mapping information, in conjunction with <code>rsimsetrtpparam</code> , is useful for creating subsequent rtP structures without compiling the block diagram.

The code generator reports a tunable fixed-point parameter according to its stored value. For example, an `sfix(16)` parameter value of 1.4 with a scaling of 2^{-8} has a value of 358 as an `int16`.

In the following example, `rsimgetrtpp` returns the parameter structure for the example model `rtwdemo_rsimtf` to `param_struct`.

```
param_struct = rsimgetrtpp('rtwdemo_rsimtf')

param_struct =

    modelChecksum: [1.7165e+009 3.0726e+009 2.6061e+009 2.3064e+009]
    parameters: [1x1 struct]
```

Save the Parameter Structure to a MAT-File

After you issue a call to `rsimgetrtpp`, save the return value of the function call to a MAT-file. Using a command-line option, you can then specify that MAT-file as input for rapid simulations.

The following example saves the parameter structure returned for `rtwdemo_rsimtf` to the MAT-file `myrsimdemo.mat`.

```
save myrsimdemo.mat param_struct;
```

For information on using command-line options to specify required files, see “Run Rapid Simulations” on page 22-18.

Convert the Parameter Structure for Running Simulations on Varying Data Sets

To use rapid simulations to test changes to specific parameters, you can convert the model parameter structure to a cell array. You can then access a specific parameter set by using the `@` operator to specify the index for a specific parameter set in the file.

To convert the structure to a cell array:

- 1 Use the function `rsimgetrtpt` to get a structure containing parameter information for the example model `rtwdemo_rsimtf`. Store the structure in a variable `param_struct`.

```
param_struct = rsimgetrtpt('rtwdemo_rsimtf');
```

The `parameters` field of the structure is a substructure that contains parameter information. The `values` field of the `parameters` substructure contains the numeric values of the parameters that you can tune during execution of the simulation code.

- 2 Use the function `rsimsetrtpparam` to expand the structure so that it contains more parameter sets. In this case, create two more parameter sets (for a total of three sets).

```
param_struct = rsimsetrtpparam(param_struct,3);
```

The function converts the `parameters` field to a cell array with three elements. Each element contains information for a single parameter set. By default, the function creates the second and third elements of the cell array by copying the first element. Therefore, all of the parameter sets use the same parameter values.

- 3 Specify new values for the parameters in the second and third parameter sets.

```
param_struct.parameters{2}.values = [-150 -5000 0 4950];  
param_struct.parameters{3}.values = [-170 -5500 0 5100];
```

- 4 Save the structure containing the parameter set information to a MAT-file.

```
save rtwdemo_rsimtf.mat param_struct;
```

Alternatively, you can modify the block parameters in the model, and use `rsimgetrtp` to create multiple parameter sets:

- 1 Use the function `rsimgetrtp` to get a structure containing parameter information for the example model `rtwdemo_rsimtf`. Store the structure in a variable `param_struct`.

```
param_struct = rsimgetrtp('rtwdemo_rsimtf');
```

- 2 Use the function `rsimsetrtpparam` to expand the structure so that it contains more parameter sets. In this case, create two more parameter sets (for a total of three sets).

```
param_struct = rsimsetrtpparam(param_struct,3);
```

The function converts the `parameters` field to a cell array with three elements. Each element contains information for a single parameter set. By default, the function creates the second and third elements of the cell array by copying the first element. Therefore, all of the parameter sets use the same parameter values.

- 3 Change the values of block parameters or workspace variables. For example, change the value of the variable `w` from 70 to 72.

```
w = 72;
```

- 4 Use `rimsgetrtp` to get another structure containing parameter information. Store the structure in a temporary variable `rtp_temp`.

```
rtp_temp = rimsgetrtp('rtwdemo_rsimtf');
```

- 5 Assign the value of the `parameters` field of `rtp_temp` to the structure `param_struct` as a second parameter set.

```
param_struct.parameters{2} = rtp_temp.parameters;
```

- 6 Change the value of the variable `w` from 72 to 75.

```
w = 75;
```

- 7 Use `rimsgetrtp` to get another structure containing parameter information. Then, assign the value of the `parameters` field to `param_struct` as a third parameter set.

```
rtp_temp = rimsgetrtp('rtwdemo_rsimtf');  
param_struct.parameters{3} = rtp_temp.parameters;
```

- 8 Save the structure containing the parameter set information to a MAT-file.

```
save rtwdemo_rsimtf.mat param_struct;
```

For more information on how to specify each parameter set when you run the simulations, see “Change Block Parameters for an RSim Simulation” on page 22-27.

Create a MAT-File for a From File Block

You can use a MAT-file as the input data source for a **From File** block. The format of the data in the MAT-file must match the data format expected by that block. For example, if you are using a matrix as an input for the MAT file, this cannot be different from the matrix size for the executable.

To create a MAT-file for a From File block:

- 1 For array format data, in the workspace create a matrix that consists of two or more rows. The first row must contain monotonically increasing time points. Other rows contain data points that correspond to the time point in that column. The time and data points must be data of type **double**.

For example:

```
t=[0:0.1:2*pi]';  
Ina1=[2*sin(t) 2*cos(t)];  
Ina2=sin(2*t);  
Ina3=[0.5*sin(3*t) 0.5*cos(3*t)];  
var_matrix=[t Ina1 Ina2 Ina3]';
```

For other supported data types, such as **int16** or fixed-point, the time data points must be of type **double**, just as for array format data. However, the sample data can be of any dimension.

For more information on setting up the input data, see the description of the **From File** block in the Simulink documentation.

- 2 Save the matrix to a MAT-file.

The following example saves the matrix `var_matrix` to the MAT-file `myrsimdemo.mat` in Version 7.3 format.

```
save '-v7.3' myrsimdemo.mat var_matrix;
```

Using a command-line option, you can then specify that MAT-file as input for rapid simulations.

Create a MAT-File for an Inport Block

You can use a MAT-file as the input data source for an Inport block.

The format of the data in the MAT-file must adhere to one of the three column-based formats listed in the following table. The table lists the formats in order from least flexible to most flexible.

Format	Description
Single time/data matrix	<ul style="list-style-type: none"> • Least flexible. • One variable. • Two or more <i>columns</i>. Number of columns must equal the sum of the dimensions of all root Inport blocks plus 1. First column must contain monotonically increasing time points. Other columns contain data points that correspond to the time point in a given row. • Data of type <code>double</code>. <p>For an example, see Single time/data matrix in the following procedure, step 4. For more information, see “Create Data Arrays for Root-Level Inports” in the Simulink documentation.</p>
Signal-and-time structure	<ul style="list-style-type: none"> • More flexible than the single time/data matrix format. • One variable. • Must contain two top-level fields: <code>time</code> and <code>signals</code>. The <code>time</code> field contains a <i>column</i> vector of the simulation times. The <code>signals</code> field contains an array of substructures, each of which corresponds to an Inport block. The substructure index corresponds to the Inport block number. Each <code>signals</code> substructure must contain a field named <code>values</code>. The <code>values</code> field must contain an array of inputs for the corresponding Inport block, where each input corresponds to a time point specified by the <code>time</code> field. • If the <code>time</code> field is set to an empty value, clear the check box for the Inport block Interpolate data parameter. • Data type must match Inport block settings. <p>For an example, see Signal-and-time structure in the following procedure, step 4. For more information on this format, see “Create Data Structures for Root Inports” in the Simulink documentation.</p>

Format	Description
Per-port structure	<ul style="list-style-type: none"> • Most flexible • Multiple variables. Number of variables must equal the number of Inport blocks. • Consists of a separate structure-with-time or structure-without-time for each Inport block. Each Inport block data structure has only one signals field. To use this format, in the Input text field, enter the names of the structures as a comma-separated list, in1, in2, ..., inN, where in1 is the data for your model's first port, in2 for the second port, and so on. • Each variable can have a different time vector. • If the time field is set to an empty value, clear the check box for the Inport block Interpolate data parameter. • Data type must match Inport block settings. • To save multiple variables to the same data file, you must save them in the order expected by the model, using the -append option. <p>For an example, see Per-port structure in the following procedure, step 4. For more information, see “Create Data Structures for Root Inports” in the Simulink documentation.</p>

The supported formats and the following procedure are illustrated in `rtwdemo_rsim_i`.

To create a MAT-file for an Inport block:

- 1** Choose one of the preceding data file formats.
- 2** Update Inport block parameter settings and specifications to match specifications of the data to be supplied by the MAT-file.

By default, the Inport block inherits parameter settings from downstream blocks. To import data from an external MAT-file, explicitly set the following parameters to match the source data in the MAT-file.

- **Main > Interpolate data**
- **Signal Attributes > Port dimensions**
- **Signal Attributes > Data type**
- **Signal Attributes > Signal type**

If you choose to use a structure format for workspace variables and the `time` field is empty, you must clear **Interpolate data** or modify the field so that it is set to a nonempty value. Interpolation requires time data.

For descriptions of the preceding block parameters, see the description of the **Inport** block in the Simulink documentation.

- 3 Build an RSim executable program for the model. The build process creates and calculates a structural checksum for the model and embeds it in the generated executable. The RSim target uses the checksum to verify that data being passed into the model is consistent with what the model executable expects.
- 4 Create the MAT-file that provides the source data for the rapid simulations. You can create the MAT-file from a workspace variable. Using the specifications in the preceding format comparison table, create the workspace variables for your simulations.

An example of each format follows:

Single time/data matrix

```
t=[0:0.1:2*pi]';
Ina1=[2*sin(t) 2*cos(t)];
Ina2=sin(2*t);
Ina3=[0.5*sin(3*t) 0.5*cos(3*t)];
var_matrix=[t Ina1 Ina2 Ina3];
```

Signal-and-time structure

```
t=[0:0.1:2*pi]';
var_single_struct.time=t;
var_single_struct.signals(1).values(:,1)=2*sin(t);
var_single_struct.signals(1).values(:,2)=2*cos(t);
var_single_struct.signals(2).values=sin(2*t);
var_single_struct.signals(3).values(:,1)=0.5*sin(3*t);
var_single_struct.signals(3).values(:,2)=0.5*cos(3*t);
v=[var_single_struct.signals(1).values...
var_single_struct.signals(2).values...
var_single_struct.signals(3).values];
```

Per-port structure

```
t=[0:0.1:2*pi]';
Inb1.time=t;
```



```
Inb1.signals.values(:,1)=2*sin(t);
Inb1.signals.values(:,2)=2*cos(t);
t=[0:0.2:2*pi]';
Inb2.time=t;
Inb2.signals.values(:,1)=sin(2*t);
t=[0:0.1:2*pi]';
Inb3.time=t;
Inb3.signals.values(:,1)=0.5*sin(3*t);
Inb3.signals.values(:,2)=0.5*cos(3*t);
```

- 5 Save the workspace variables to a MAT-file.

Single time/data matrix

The following example saves the workspace variable `var_matrix` to the MAT-file `rsim_i_matrix.mat`.

```
save rsim_i_matrix.mat var_matrix;
```

Signal-and-time structure

The following example saves the workspace structure variable `var_single_struct` to the MAT-file `rsim_i_single_struct.mat`.

```
save rsim_i_single_struct.mat var_single_struct;
```

Per-port structure

To order data when saving per-port structure variables to a single MAT-file, use the `save` command's `-append` option. Be sure to append the data in the order that the model expects it.

The following example saves the workspace variables `Inb1`, `Inb2`, and `Inb3` to MAT-file `rsim_i_multi_struct.mat`.

```
save rsim_i_multi_struct.mat Inb1;
save rsim_i_multi_struct.mat Inb2 -append;
save rsim_i_multi_struct.mat Inb3 -append;
```

The `save` command does not preserve the order in which you specify your workspace variables in the command line when saving data to a MAT-file. For example, if you specify the variables `v1`, `v2`, and `v3`, in that order, the order of the variables in the MAT-file could be `v2 v1 v3`.

Using a command-line option, you can then specify the MAT-files as input for rapid simulations.

Scripts for Batch and Monte Carlo Simulations

The RSim target is for batch simulations in which parameters and input signals vary for multiple simulations. New output file names allow you to run new simulations without overwriting prior simulation results. You can set up a series of simulations to run by creating a `.bat` file for use on a Microsoft Windows platform.

Create a file for the Windows platform with a text editor and execute it by typing the file name, for example, `mybatch`, where the name of the text file is `mybatch.bat`.

```
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run1.mat -o results1.mat -tf 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run2.mat -o results2.mat -tf 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run3.mat -o results3.mat -tf 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run4.mat -o results4.mat -tf 10.0
```

In this case, batch simulations run using four sets of input data in files `run1.mat`, `run2.mat`, and so on. The RSim executable saves the data to the files specified with the `-o` option.

The variable names containing simulation results in each of the files are identical. Therefore, loading consecutive sets of data without renaming the data once it is in the MATLAB workspace results in overwriting the prior workspace variable with new data. To avoid overwriting, you can copy the result to a new MATLAB variable before loading the next set of data.

You can also write MATLAB scripts to create new signals and new parameter structures, as well as to save data and perform batch runs using the bang command (!).

For details on running simulations and available command-line options, see “Run Rapid Simulations” on page 22-18. For an example of a rapid simulation batch script, see the example “Run Batch Simulations Without Recompiling Generated Code”.

Run Rapid Simulations

- “Rapid Simulations” on page 22-19
- “Requirements for Running Rapid Simulations” on page 22-20
- “Set a Clock Time Limit for a Rapid Simulation” on page 22-21
- “Override a Model Simulation Stop Time” on page 22-21

- “Read the Parameter Vector into a Rapid Simulation” on page 22-22
- “Specify New Signal Data File for a From File Block” on page 22-22
- “Specify Signal Data File for an Inport Block” on page 22-25
- “Change Block Parameters for an RSim Simulation” on page 22-27
- “Specify a New Output File Name for a Simulation” on page 22-29
- “Specify New Output File Names for To File Blocks” on page 22-29

Rapid Simulations

Using the RSim target, you can build a model once and run multiple simulations to study effects of varying parameter settings and input signals. You can run a simulation directly from your operating system command line, redirect the command from the MATLAB command line by using the bang (!) character, or execute commands from a script.

From the operating system command line, use

```
rtwdemo_rsimgtf
```

From the MATLAB command line, use

```
!rtwdemo_rsimgtf
```

The following table lists ways you can use RSim target command-line options to control a simulation.

To...	Use...
Read input data for a From File block from a MAT-file other than the MAT-file used for the previous simulation	<code>model -f oldfilename.mat=newfilename.mat</code>
Print a summary of the options for RSim executable targets	<code>executable filename -h</code>
Read input data for an Inport block from a MAT-file	<code>model -i filename.mat</code>
Time out after n clock time seconds, where n is a positive integer value	<code>model -L n</code>
Write MAT-file logging data to file <code>filename.mat</code>	<code>model -o filename.mat</code>
Read a parameter vector from file <code>filename.mat</code>	<code>model -p filename.mat</code>

To...	Use...
Override the default TCP port (17725) for external mode	<code>model -port TCPport</code>
Write MAT-file logging data to a MAT-file other than the MAT-file used for the previous simulation	<code>model -t oldfilename.mat=newfilename.mat</code>
Run the simulation until the time value <i>stoptime</i> is reached	<code>model -tf stoptime</code>
Run in verbose mode	<code>model -v</code>
Wait for the Simulink engine to start the model in external mode	<code>model -w</code>

The following sections use the `rtwdemo_rsimtf` example model in examples to illustrate some of these command-line options. In each case, the example assumes you have already done the following:

- Created or changed to a working folder.
- Opened the example model.
- Copied the data file `matlabroot/toolbox/rtw/rtwdemos/rsimdemos/rsim_tfdata.mat` to your working folder. You can perform this operation using the command:

```
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos',...
'rsimdemos','rsim_tfdata.mat'),pwd);
```

Requirements for Running Rapid Simulations

The following requirements apply to both fixed and variable step executables.

- You must run the RSim executable on a computer configured to run MATLAB. Also, the `RSim.exe` file must be able to access the MATLAB and Simulink installation folders on this machine. To obtain that access, your `PATH` environment variable must include `/bin` and `/bin/($ARCH)`, where `($ARCH)` represents your operating system architecture. For example, for a personal computer running on a Windows platform, `($ARCH)` is “win64”, whereas for a Linux machine, `($ARCH)` is “glnxa64”.
- On GNU Linux platforms, to run an RSim executable, define the `LD_LIBRARY_PATH` environment variable to provide the path to the MATLAB installation folder, as follows:

```
% setenv LD_LIBRARY_PATH /matlab/sys/os/glnx64:$LD_LIBRARY_PATH
```

- On the Apple Macintosh OS X platform, to run RSim target executables, you must define the environment variable `DYLD_LIBRARY_PATH` to include the folders `bin/mac` and `sys/os/mac` under the MATLAB installation folder. For example, if your MATLAB installation is under `/MATLAB`, add `/MATLAB/bin/mac` and `/MATLAB/sys/os/mac` to the definition for `DYLD_LIBRARY_PATH`.

Set a Clock Time Limit for a Rapid Simulation

If a model experiences frequent zero crossings and the model's minor step size is small, consider setting a time limit for a rapid simulation. To set a time limit, specify the `-L` option with a positive integer value. The simulation aborts after running for the specified amount of clock time (not simulation time). For example,

```
!rtwdemo_rsimgtf -L 20
```

Based on your clock, after the executable runs for 20 seconds, the program terminates. You see a message similar to one of the following:

- On a Microsoft Windows platform,


```
Exiting program, time limit exceeded
Logging available data ...
```
- On The Open Group UNIX platform,


```
** Received SIGALRM (Alarm) signal @ Fri Jul 25 15:43:23 2003
** Exiting model 'vdp' @ Fri Jul 25 15:43:23 2003
```

You do not need to do anything to your model or to its configuration to use this option.

Override a Model Simulation Stop Time

By default, a rapid simulation runs until the simulation time reaches the time specified the Configuration Parameters dialog box, on the **Solver** pane. You can override the model simulation stop time by using the `-tf` option. For example, the following simulation runs until the time reaches 6.0 seconds.

```
!rtwdemo_rsimgtf -tf 6.0
```

The RSim target stops and logs output data using MAT-file data logging rules.

If the model includes a From File block, the end of the simulation is regulated by the stop time setting specified in the Configuration Parameters dialog box, on the **Solver** pane, or with the RSim target option `-tf`. The values in the block's time vector are ignored.

However, if the simulation time exceeds the endpoints of the time and signal matrix (if the final time is greater than the final time value of the data matrix), the signal data is extrapolated to the final time value.

Read the Parameter Vector into a Rapid Simulation

To read the model parameter vector into a rapid simulation, you must first create a MAT-file that includes the parameter structure as described in “Create a MAT-File That Includes a Model Parameter Structure” on page 22-9. You can then specify the MAT-file in the command line with the `-p` option.

For example:

- 1 Build an RSim executable for the example model `rtwdemo_rsimtf`.
- 2 Modify parameters in your model and save the parameter structure.

```
param_struct = rsimgetrtp('rtwdemo_rsimtf');  
save myrsimdata.mat param_struct
```

- 3 Run the executable with the new parameter set.

```
!rtwdemo_rsimtf -p myrsimdata.mat
```

```
** Starting model 'rtwdemo_rsimtf' @ Tue Dec 27 12:30:16 2005  
** created rtwdemo_rsimtf.mat **
```

- 4 Load workspace variables and plot the simulation results by entering the following commands:

```
load myrsimdata.mat  
plot(rt_yout)
```

Specify New Signal Data File for a From File Block

If your model's input data source is a `From File` block, you can feed the block with input data during simulation from a single MAT-file or you can change the MAT-file from one simulation to the next. Each MAT-file must adhere to the format described in “Create a MAT-File for a From File Block” on page 22-13.

To change the MAT-file after an initial simulation, you specify the executable with the `-f` option and an `oldfile.mat=newfile.mat` parameter, as shown in the following example.

- 1 Set some parameters in the MATLAB workspace. For example:

```
w = 100;
```

```
theta = 0.5;
```

- 2 Build an RSim executable for the example model `rtwdemo_rsimtf`.
- 3 Run the executable.

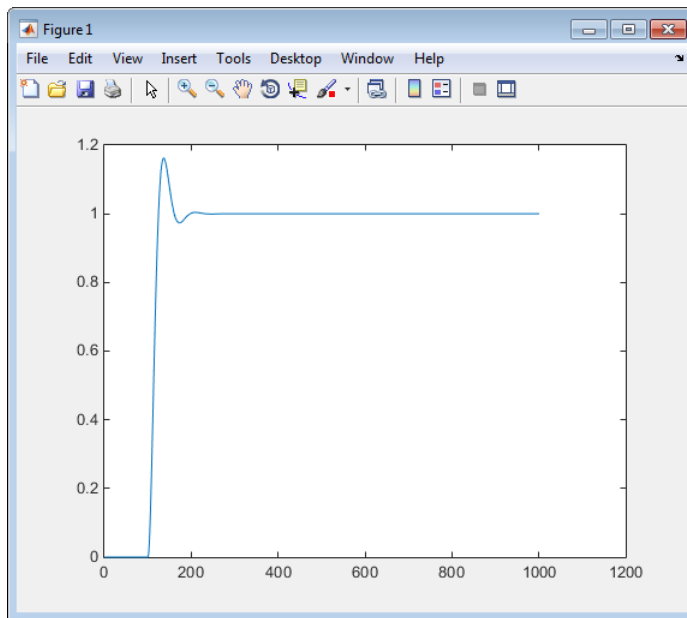
```
!rtwdemo_rsimtf
```

The RSim executable runs a set of simulations and creates output MAT-files containing the specific simulation result.

- 4 Load the workspace variables and plot the simulation results by entering the following commands:

```
load rtwdemo_rsimtf.mat  
plot(rt_yout)
```

The resulting plot shows simulation results based on default input data.



- 5 Create a new data file, `newfrom.mat`, that includes the following data:

```
t=[0:.001:1];  
u=sin(100*t.*t);  
tu=[t;u];
```

```
save newfrom.mat tu;
```

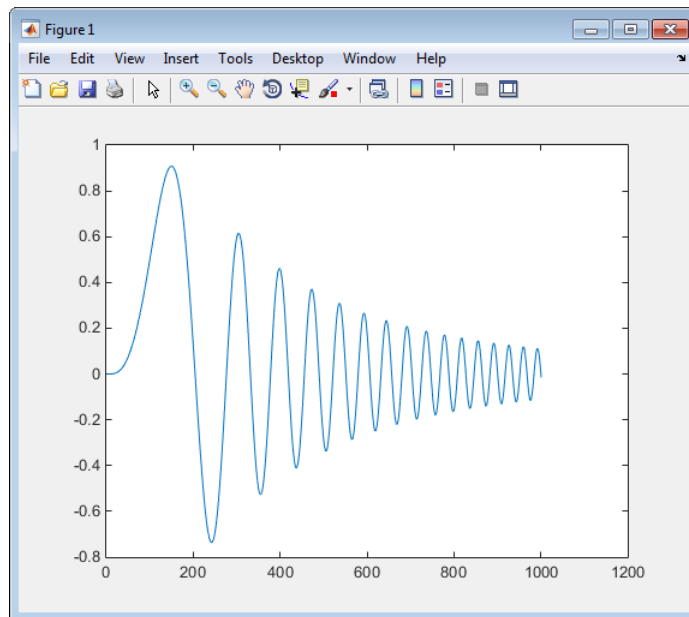
- 6 Run a rapid simulation with the new data by using the `-f` option to replace the original file, `rsim_tfdata.mat`, with `newfrom.mat`.

```
!rtwdemo_rsimtf -f rsim_tfdata.mat=newfrom.mat
```

- 7 Load the data and plot the new results by entering the following commands:

```
load rtwdemo_rsimtf.mat
plot(rt_yout)
```

The next figure shows the resulting plot.



From File blocks require input data of type **double**. If you need to import signal data of a data type other than **double**, use an Inport block (see “Create a MAT-File for an Inport Block” on page 22-14) or a From Workspace block with the data specified as a structure.

Workspace data must be in the format:

```
variable.time
variable.signals.values
```


If you have more than one signal, use the following format:

```
variable.time  
variable.signals(1).values  
variable.signals(2).values
```

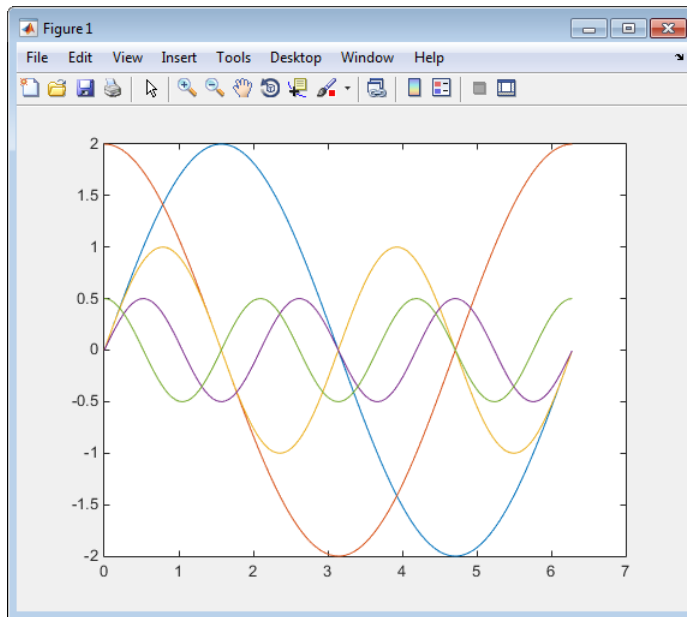
Specify Signal Data File for an Inport Block

If your model's input data source is an Inport block, you can feed the block with input data during simulation from a single MAT-file or you can change the MAT-file from one simulation to the next. Each MAT-file must adhere to one of the three formats described in “Create a MAT-File for an Inport Block” on page 22-14.

To specify the MAT-file after a simulation, you specify the executable with the `-i` option and the name of the MAT-file that contains the input data. For example:

- 1 Open the model `rtwdemo_rsim_i`.
- 2 Check the Inport block parameter settings. The following Inport block data parameter settings and specifications that you specify for the workspace variables must match settings in the MAT-file, as indicated in “Configure Inports to Provide Simulation Source Data” on page 22-6:
 - **Main > Interpolate data**
 - **Signal Attributes > Port dimensions**
 - **Signal Attributes > Data type**
 - **Signal Attributes > Signal type**
- 3 Build the model.
- 4 Set up the input signals. For example:

```
t=[0:0.01:2*pi]';  
s1=[2*sin(t) 2*cos(t)];  
s2=sin(2*t);  
s3=[0.5*sin(3*t) 0.5*cos(3*t)];  
plot(t, [s1 s2 s3])
```



- 5 Prepare the MAT-file by using one of the three available file formats described in “Create a MAT-File for an Inport Block” on page 22-14. The following example defines a signal-and-time structure in the workspace and names it `var_single_struct`.

```
t=[0:0.1:2*pi]';
var_single_struct.time=t;
var_single_struct.signals(1).values(:,1)=2*sin(t);
var_single_struct.signals(1).values(:,2)=2*cos(t);
var_single_struct.signals(2).values=sin(2*t);
var_single_struct.signals(3).values(:,1)=0.5*sin(3*t);
var_single_struct.signals(3).values(:,2)=0.5*cos(3*t);
v=[var_single_struct.signals(1).values...
var_single_struct.signals(2).values...
var_single_struct.signals(3).values];
```

- 6 Save the workspace variable `var_single_struct` to MAT-file `rsim_i_single_struct`.

```
save rsim_i_single_struct.mat var_single_struct;
```

- 7 Run a rapid simulation with the input data by using the `-i` option. Load and plot the results.

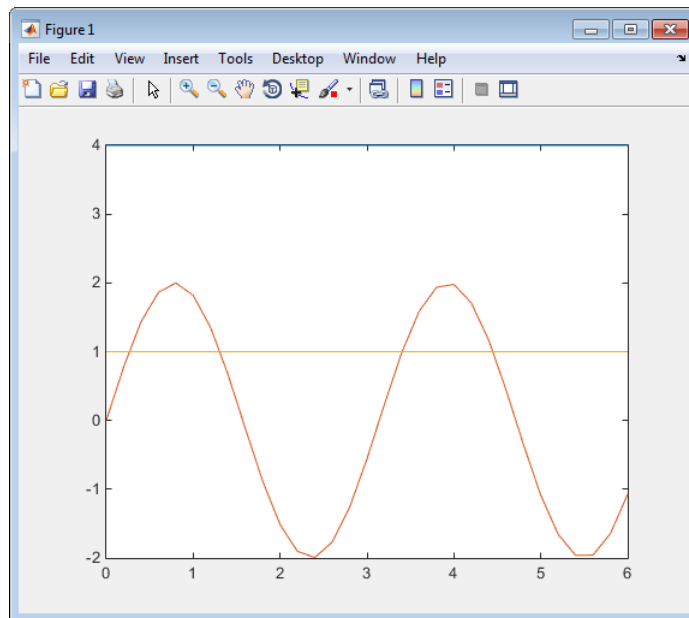
```
!rtwdemo_rsim_i -i rsim_i_single_struct.mat

** Starting model 'rtwdemo_rsim_i' @ Tue Aug 19 10:26:53 2014
*** rsim_i_single_struct.mat is successfully loaded! ***
** created rtwdemo_rsim_i.mat **

** Execution time = 0.02024185130718954s
```

- 8 Load and plot the results.

```
load rtwdemo_rsim_i.mat
plot(rt_tout, rt_yout);
```



Change Block Parameters for an RSim Simulation

As described in “Create a MAT-File That Includes a Model Parameter Structure” on page 22-9, after you alter one or more parameters in a Simulink block diagram, you can extract the parameter vector, `model_P`, for the entire model. You can then save the parameter vector, along with a model checksum, to a MAT-file. This MAT-file can be read directly by the standalone RSim executable, allowing you to replace the entire parameter

vector or individual parameter values, for running studies of variations of parameter values representing coefficients, new data for input signals, and so on.

RSim can read the MAT-file and replace the entire *model_P* structure whenever you change one or more parameters, without recompiling the entire model.

For example, assume that you changed one or more parameters in your model, generated the new *model_P* vector, and saved *model_P* to a new MAT-file called *mymatfile.mat*. To run the same *rtwdemo_rsimtf* model and use these new parameter values, use the `-p` option, as shown in the following example:

```
!rtwdemo_rsimtf -p mymatfile.mat
load rtwdemo_rsimtf
plot(rt_yout)
```

If you have converted the parameter structure to a cell array for running simulations on varying data sets, as described in “Convert the Parameter Structure for Running Simulations on Varying Data Sets” on page 22-11, you must add an *@n* suffix to the MAT-file specification. *n* is the element of the cell array that contains the specific input that you want to use for the simulation.

The following example converts *param_struct* to a cell array, changes parameter values, saves the changes to MAT-file *mymatfile.mat*, and then runs the executable using the parameter values in the second element of the cell array as input.

```
param_struct = rsimgetrtp('rtwdemo_rsimtf');
param_struct = rsimsetrtpparam(param_struct,2);
param_struct.parameters{1}

ans =

    dataTypeName: 'double'
    dataTypeId: 0
    complex: 0
    dtTransIdx: 0
    values: [-140 -4900 0 4900]
    map: []
    structParamInfo: []

param_struct.parameters{2}.values=[-150 -5000 0 4950];
save mymatfile.mat param_struct;
!rtwdemo_rsimtf -p mymatfile.mat@2 -o rsim2.mat
```

Specify a New Output File Name for a Simulation

If you have specified one or more of the **Save to Workspace** options — **Time**, **States**, **Outputs**, or **Final States** — in the Configuration Parameters dialog box, on the **Data Import/Export** pane, the default is to save simulation logging results to the file `model.mat`. For example, the example model `rtwdemo_rsimtf` normally saves data to `rtwdemo_rsimtf.mat`, as follows:

```
!rtwdemo_rsimtf
created rtwdemo_rsimtf.mat
```

You can specify a new output file name for data logging by using the `-o` option when you run an executable.

```
!rtwdemo_rsimtf -o rsim1.mat
```

In this case, the set of parameters provided at the time of code generation, including From File block data parameters, is run.

Specify New Output File Names for To File Blocks

In much the same way as you can specify a new system output file name, you can also provide new output file names for data saved from one or more To File blocks. To do this, specify the original file name at the time of code generation with a new name, as shown in the following example:

```
!rtwdemo_rsimtf -t rtwdemo_rsimtf_data.mat=mynewsimdata.mat
```

In this case, assume that the original model wrote data to the output file `rtwdemo_rsimtf_data.mat`. Specifying a new file name forces RSim to write to the file `mynewsimdata.mat`. With this technique, you can avoid overwriting an existing simulation run.

Tune Parameters Interactively During Rapid Simulation

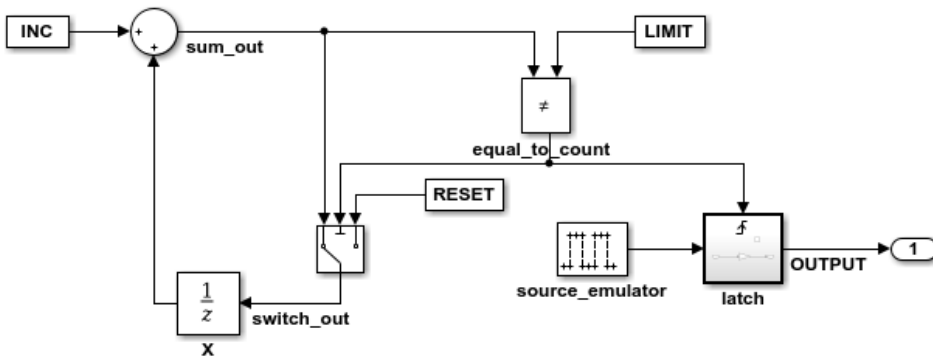
The RSim target was designed to let you run batch simulations at the fastest possible speed. Using variable-step or fixed-step solvers with RSim combined with the use of a tunable parameter data structure, whether you set **Default parameter behavior** to **Tunable** or to **Inlined**, you can create multiple parameter sets to run with the RSim target's standalone executable file (.exe on Windows) generated using Simulink Coder. Each invocation of the executable allows specification of the file name to use for results.

For this example, **Default parameter behavior** is set to **Inlined**. The model declares workspace variables as tunable parameters. To use RSim with **Default parameter behavior** set to **Tunable**, and without explicitly declaring tunable parameters, see “Run Batch Simulations Without Recompiling Generated Code”.

Open Example Model

Open the example model `rtwdemo_rsim_param_tuning`.

```
open_system('rtwdemo_rsim_param_tuning');
```



1. Build Model with RSim Target
2. Run RSIMGETRTP
3. Save RTP structure in a MAT-File
4. Open the MATLAB RSim GUI Example

Copyright 1994-2012 The MathWorks, Inc.

The RSim target was designed to let you run batch simulations at the fastest possible speed. Using variable-step or fixed-step solvers with RSim combined with the use of a tunable parameter data structure, you can create multiple parameter sets to run with the RSim target's standalone executable file (.exe on Windows) generated using Simulink Coder. Each invocation of the executable allows specification of the filename to use for results.

This model uses the RSim target to allow a non-real-time executable to be passed new data without the need to recompile the Simulink model. This feature allows you to easily get a map of the tunable parameters declared in a model and save it in a MAT-file. You can then create your own MATLAB GUI or a standalone GUI (independent of MATLAB) to read and write the MAT-file and rerun the executable to produce new output files.

View MATLAB programs

Double-click the buttons at the upper right sequentially to run the example. To review the code used to create both the MATLAB GUI and standalone GUI, double-click the View MATLAB programs button.

For more information, you can also refer to the "Rapid Simulation Target" section in the Simulink Coder documentation.

This model uses the RSim target and the `rsimgetrtpt` function to allow a non real time executable to be passed new data without the need to recompile the Simulink model. This feature allows you to get a map of the tunable parameters declared in a model and save it in a MAT-file. You can then create your own MATLAB GUI or a standalone GUI (independent of MATLAB) to read and write the MAT-file and rerun the executable to produce new output files.

Double-click the buttons at the upper right sequentially to run the example.

To review the code used to create both the MATLAB GUI and standalone GUI, double-click the View MATLAB programs button.

For more information, you can also refer to the "Rapid Simulation Target" section in the Simulink Coder documentation.

Rapid Simulation Target Limitations

The RSim target has the following limitations:

- Does not support algebraic loops.
- Does not support Interpreted MATLAB Function blocks.
- Does not support noninlined MATLAB language or Fortran S-functions.
- If an RSim build includes referenced models (by using Model blocks), set up these models to use fixed-step solvers to generate code for them. The top model, however, can use a variable-step solver as long as the blocks in the referenced models are discrete.
- In certain cases, changing block parameters can result in structural changes to your model that change the model checksum. An example of such a change is changing the number of delays in a DSP simulation. In such cases, you must regenerate the code for the model.

More About

- “Acceleration”
- “Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target” on page 22-33

Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target

S-functions are an important class of system target file for which the code generator can produce code. The ability to encapsulate a subsystem into an S-function allows you to increase its execution efficiency and facilitate code reuse.

The following sections describe the properties of S-function targets and illustrate how to generate them. For more details on the structure of S-functions, see “Host-Specific Code”.

In this section...

- “About the S-Function Target” on page 22-33
- “Create S-Function Blocks from a Subsystem” on page 22-36
- “Tunable Parameters in Generated S-Functions” on page 22-40
- “System Target File” on page 22-42
- “Checksums and the S-Function Target” on page 22-42
- “Generated S-Function Compatibility” on page 22-43
- “S-Function Target Limitations” on page 22-43

About the S-Function Target

Using the S-function target, you can build an S-function component and use it as an S-Function block in another model. The following sections describe deployment considerations for the S-function target.

- “Required Files for S-Function Deployment” on page 22-34
- “Sample Time Propagation in Generated S-Functions” on page 22-35
- “Choose a Solver Type” on page 22-35
- “Solver Type Overrides” on page 22-36

The ‘S-Function’ value for CodeFormat TLC variable used by the S-function target generates code that conforms to the Simulink C MEX S-function application programming interface (API). Applications of this format include

- Conversion of a model to a component. You can generate an S-Function block for a model, `m1`. Then, you can place the generated S-Function block in another model, `m2`. Regenerating code for `m2` does not require regenerating code for `m1`.

- Conversion of a subsystem to a component. By extracting a subsystem to a separate model and generating an S-Function block from that model, you can create a reusable component from the subsystem. See “Create S-Function Blocks from a Subsystem” on page 22-36 for an example of this procedure.
- Speeding up simulation. Often, an S-function generated from a model performs more efficiently than the original model.
- Code reuse. You can incorporate multiple instances of one model inside another without replicating the code for each instance. Each instance continues to maintain its own unique data.

You can place a generated S-function block into another model from which you can generate another S-function. This approach allows any level of nested S-functions. For limitations related to nesting, see “Limitations on Nesting S-Functions” on page 22-47.

Note: While the S-function target provides a means to deploy an application component for reuse while shielding its internal logic from inspection and modification, the preferred solutions for protecting intellectual property in distributed components are:

- The protected model, a referenced model that hides all block and line information. For more information, see “Protected Model” in the Simulink documentation.
 - The shared library system target file, used to generate a shared library for a model or subsystem for use in a system simulation external to Simulink. For more information, see “Package Generated Code as Shared Object Libraries” in the Embedded Coder documentation.
-

Required Files for S-Function Deployment

There are different files required to deploy a generated S-Function block for simulation versus code generation.

To deploy your generated S-Function block for inclusion in other models *for simulation*, you need only provide the binary MEX-file object that was generated in the current working folder when the S-Function block was created. The required file is:

- `subsys_sf.mexext`

where `subsys` is the subsystem name and `mexext` is a platform-dependent MEX-file extension (see `mexext`). For example, `SourceSubsys_sf.mexw64`.

To deploy your generated S-Function block for inclusion in other models *for code generation*, you must provide all of the files that were generated in the current working folder when the S-Function block was created. The required files are:

- `subsys_sf.c` or `.cpp`, where `subsys` is the subsystem name (for example, `SourceSubsys_sf.c`)
- `subsys_sf.h`
- `subsys_sf.mexext`, where `mexext` is a platform-dependent MEX-file extension (see `mexext`)
- Subfolder `subsys_sfcn_rtw` and its contents

Note: The generated S-function code uses **Configuration Parameters > Hardware Implementation** parameter values that match the host system on which the function was built. When you use the S-function in a model for code generation, make sure that these parameter values for the model match the parameter values of the S-function.

Sample Time Propagation in Generated S-Functions

A generated S-Function block can inherit its sample time from the model in which it is placed if certain criteria are met. Conditions that govern sample time propagation for both Model blocks and generated S-Function blocks are described in “Sample Times for Model Referencing” in the Simulink documentation and “Inherited Sample Time for Referenced Models” on page 8-25.

To generate an S-Function block that meets the criteria for inheriting sample time, you must constrain the solver for the model from which the S-Function block is generated. On the **Solver** configuration parameters dialog box pane, set **Type** to **Fixed-step** and **Periodic sample time constraint** to **Ensure sample time independent**. If the model is unable to inherit sample times, this setting causes the Simulink software to display an error message when building the model. See “Periodic sample time constraint” in the Simulink documentation for more information about this option.

Choose a Solver Type

The table shows the possible combinations of top-level model solver types as these types relate to subsystem build types and solver types for generated S-functions.

Top-level Model Solver Options and Subsystem Build Types

	Model Configuration Parameters: top-level model configuration	
Subsystem Build Type	Solver options, Type: Variable-step	Solver options, Type: Fixed-step
Build This Subsystem	Generated S-function does NOT require variable-step solver	Generated S-function does NOT require variable-step solver
Generate S-Function	Generated S-function requires variable-step solver	Generated S-function does NOT require variable-step solver

Note: S-functions generated from a subsystem have parameters that are hard coded into the block. Simulink calculates parameters such as sample time when it generates the block, not during simulation run time. Hence, it is important to verify whether the generated S-Function block works as expected in the destination model.

Solver Type Overrides

There are instances when the subsystem build type selection produces an override of the subsystem solver type. The table summarizes the relationships between subsystem build types and the applied subsystem solver types.

Top-level Model Solver Type Overrides of Subsystem Solver Types by Build Type

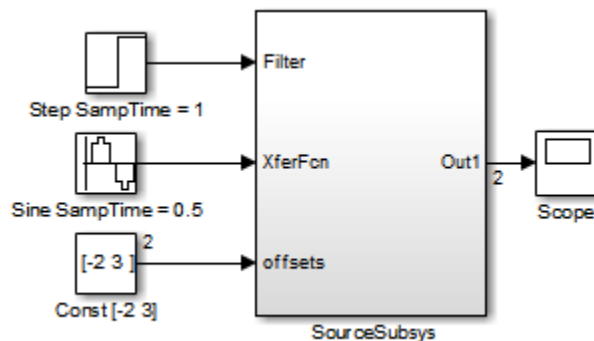
	Model Configuration Parameters: top-level model configuration	
Subsystem Build Type	Solver options, Type: Variable-step	Solver options, Type: Fixed-step
Build This Subsystem	Subsystem uses fixed-step solver type	Subsystem uses fixed-step solver type
Generate S-Function	Subsystem uses variable-step solver type	Subsystem uses fixed-step solver type

Create S-Function Blocks from a Subsystem

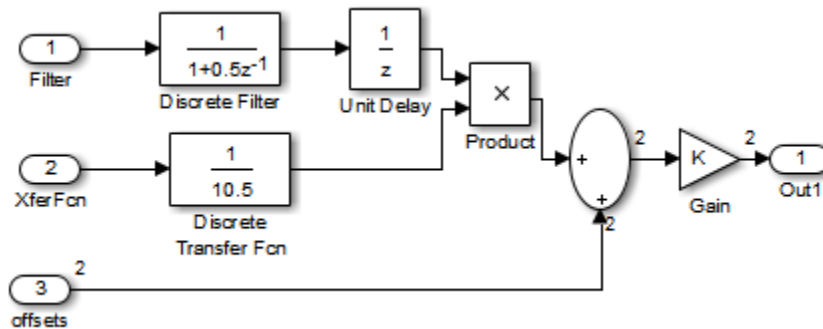
This section illustrates how to extract a subsystem from a model and generate a reusable S-function component from it.

The next figure shows **SourceModel**, a simple model that inputs signals to a subsystem. The subsequent figure shows the subsystem, **SourceSubsys**. The signals, which have different widths and sample times, are:

- A Step block with sample time 1
- A Sine Wave block with sample time 0.5
- A Constant block whose value is the vector [-2 3]



SourceModel



SourceSubsys

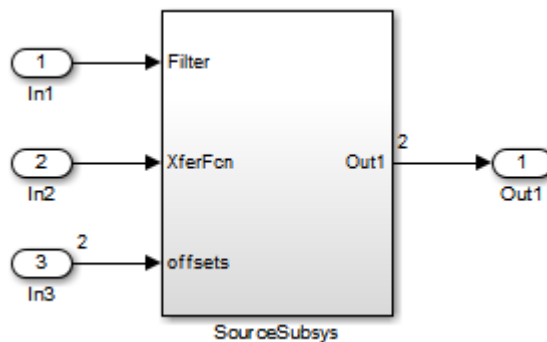
The objective is to extract `SourceSubsys` from the model and build an S-Function block from it, using the S-function target. The S-Function block must perform identically to the subsystem from which it was generated.

In this model, `SourceSubsys` inherits sample times and signal widths from its input signals. However, S-Function blocks created from a model using the S-function target has all signal attributes (such as signal widths or sample times) hard-wired. (The sole exception to this rule concerns sample times, as described in “Sample Time Propagation in Generated S-Functions” on page 22-35.)

In this example, you want the S-Function block to retain the properties of `SourceSubsys` as it exists in `SourceModel`. Therefore, before you build the subsystem as a separate S-function component, you must set the inport sample times and widths explicitly. In addition, the solver parameters of the S-function component must be the same as those parameters of the original model. The generated S-function component operates identically to the original subsystem (see “Choose a Solver Type” on page 22-35 for more information).

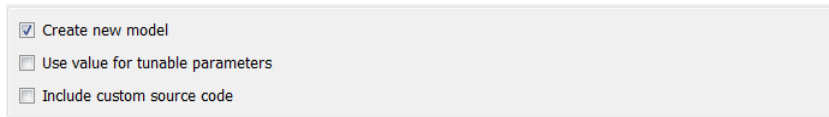
To build `SourceSubsys` as an S-function component,

- 1 Create a new model and copy/paste the `SourceSubsys` block into the empty window.
- 2 Set the signal widths and sample times of inports inside `SourceSubsys` such that they match those of the signals in the original model. Inport 1, `Filter`, has a width of 1 and a sample time of 1. Inport 2, `XferFcn`, has a width of 1 and a sample time of 0.5. Inport 3, `offsets`, has a width of 2 and a sample time of 0.5.
- 3 The generated S-Function block should have three inports and one output. Connect inports and an output to `SourceSubsys`, as shown in the next figure.



The signal widths and sample times are propagated to these ports.

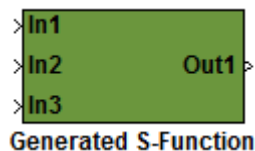
- 4 Set the solver type, mode, and other solver parameters such that they are identical to those of the source model. This is easiest to do if you use Model Explorer.
- 5 In the Configuration Parameters dialog box, go to the **Code Generation** pane.
- 6 Click **Browse** to open the System Target File Browser.
- 7 In the System Target File Browser, select the S-function target, `rtwsfcn.tlc`, and click **OK**.
- 8 Select the **S-Function Target** pane. Make sure that **Create new model** is selected, as shown in the next figure:



When this option is selected, the build process creates a new model after it builds the S-function component. The new model contains an S-Function block, linked to the S-function component.

Click **Apply**.

- 9 Save the new model containing your subsystem, for example as `SourceSubsys`.
- 10 Build the model.
- 11 The build process produces the S-function component in the working folder. After the build, a new model window is displayed.

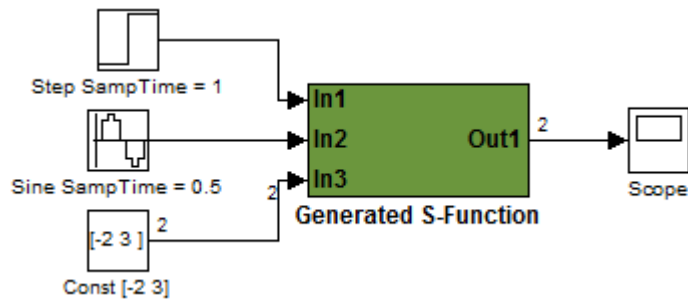


Optionally you can save the generated model, for example as `SourceSubsys_Sfunction`.

- 12 You can now copy the S-Function block generated from the new model and use it in other models or in a library.

Note: For a list of files required to deploy your S-Function block for simulation or code generation, see “Required Files for S-Function Deployment” on page 22-34.

The next figure shows the S-Function block plugged into the original model. Given identical input signals, the S-Function block performs identically to the original subsystem.



Generated S-Function Configured Like SourceModel

The speed at which the S-Function block executes is typically faster than the original model. This difference in speed is more pronounced for larger and more complicated models. By using generated S-functions, you can increase the efficiency of your modeling process.

Tunable Parameters in Generated S-Functions

You can use tunable parameters in generated S-functions in two ways:

- Use the **Generate S-function** feature (see “Automate S-Function Generation with S-Function Builder” on page 24-22).

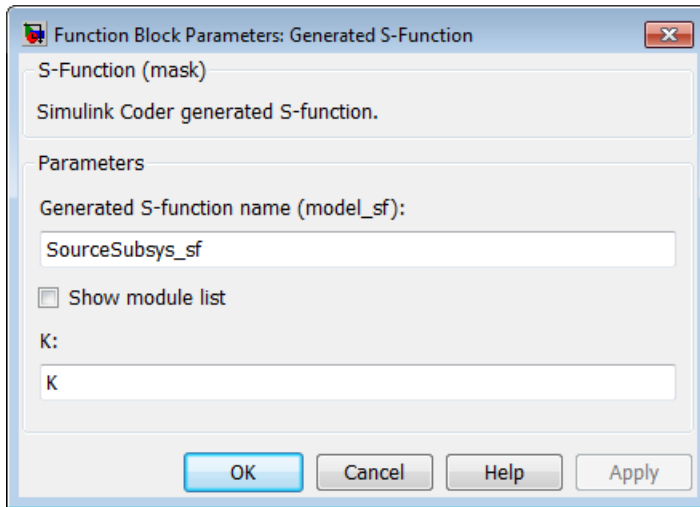
or

- Use the Model Parameter Configuration dialog box (see “Block Parameter Representation in the Generated Code” on page 14-43) to declare desired block parameters tunable.

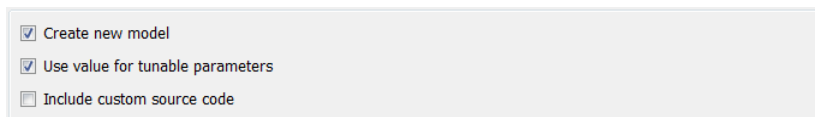
Block parameters that are declared tunable with the `auto` storage class in the source model become tunable parameters of the generated S-function. These parameters do not become part of a generated `model_P` (formerly `rtP`) parameter data structure, as they would in code generated from other targets. Instead, the generated code accesses these parameters by using MEX API calls such as `mxGetPr` or `mxGetData`. Your code should access these parameters in the same way.

For more information on MEX API calls, see “About C S-Functions” and “MATLAB API for Other Languages”.

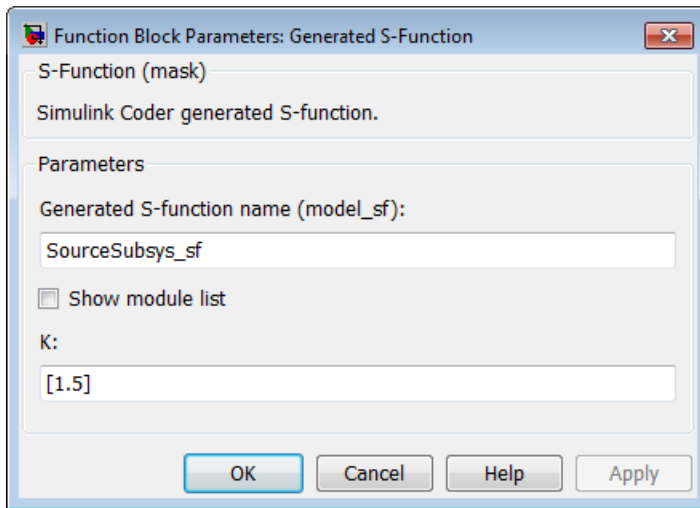
S-Function blocks created by using the S-function target are automatically masked. The mask displays each tunable parameter in an edit field. By default, the edit field displays the parameter by variable name, as in the following example.



You can choose to display the value of the parameter rather than its variable name by selecting **Use value for tunable parameters** on the **Code Generation > S-Function Target** pane of the Configuration Parameters dialog box.



When this option is chosen, the value of the variable (at code generation time) is displayed in the edit field, as in the following example.



System Target File

The `rtwsfcn.tlc` system target file is provided for use with the S-function target.

Checksums and the S-Function Target

The code generator creates a checksum for a model and uses the checksum during the build process for code reuse, model reference, and external mode features.

The code generator calculates a model checksum by

- 1 Calculating a checksum for each subsystem in the model. A subsystem's checksum is the combination of properties (data type, complexity, sample time, port dimensions, and so forth) of the subsystem's blocks.
- 2 Combining the subsystem checksums and other model-level information.

An S-function can add additional information, not captured during the block property analysis, to a checksum by calling the function `ssSetChecksumVal`. For the S-Function target, the value that gets added to the checksum is the checksum of the model or subsystem from which the S-function is generated.

The code generator applies the subsystem and model checksums as follows:

- Code reuse — If two subsystems in a model have the same checksum, the code generator produces code for one function only.
- Model reference — If the current model checksum matches the checksum when the model was built, the build process does not rebuild referenced models.
- External mode — If the current model checksum does not match the checksum of the code that is running on the target hardware, the build process generates an error.

Generated S-Function Compatibility

When you build a MEX S-function from your model, the code generator builds a level 2 noninlined S-function. Cross-release usage limitations on the generated code and binary MEX file (for example, *.mexw64) include:

- S-function target generated code from previous MATLAB release software is not compatible with newer releases. Do not recompile the generated code from a previous release with newer MATLAB release software. Use the same MATLAB release software to generate code for the S-function target and compile the code into a MEX file.
- You can use binary S-function MEX files generated from previous MATLAB release software with the same or newer releases with the same compatibility considerations as handwritten S-functions. For more information, see “S-Function Compatibility”.
- The code generator can generate code and build an executable from a model that contains generated S-functions. This support requires that the S-functions are built with the same MATLAB release software that builds the model. It is not possible to incorporate a generated S-function MEX file from previous MATLAB release software into a model and build the model with newer releases.

S-Function Target Limitations

- “Limitations on Using Tunable Variables in Expressions” on page 22-44
- “Parameter Tuning” on page 22-44
- “Run-Time Parameters and S-Function Compatibility Diagnostics” on page 22-44
- “Limitations on Using Goto and From Block” on page 22-45
- “Limitations on Building and Updating S-Functions” on page 22-46
- “Unsupported Blocks” on page 22-47
- “SimState Not Supported for Code Generation” on page 22-47

- “Profiling Code Performance Not Supported” on page 22-47
- “Limitations on Nesting S-Functions” on page 22-47
- “Limitations on User-Defined Data Types” on page 22-48
- “Limitation on Right-Click Generation of an S-Function Target” on page 22-48
- “Limitation on S-Functions with Bus I/O Signals” on page 22-48
- “Limitation on Subsystems with Function-Call I/O Signals” on page 22-49
- “Data Store Access” on page 22-49
- “Cannot Specify Outport Dimensions Through Subsystem Mask” on page 22-49

Limitations on Using Tunable Variables in Expressions

Certain limitations apply to the use of tunable variables in expressions. When the code generator encounters an unsupported expression while producing code, a warning appears and the equivalent numeric value is generated in the code. For a list of the limitations, see “Tunable Expression Limitations” on page 14-49.

Parameter Tuning

The S-Function block does not support tuning of tunable parameters with:

- Complex values.
- Values or data types that are transformed to a constant (by setting the model configuration parameter **Optimization > Signals and Parameters > Default parameter behavior** to **Inlined**).
- Data types that are not built-in.

If you select these tunable parameters (through the Generate S-Function for Subsystem dialog box):

- The software produces warnings during the build process.
- The generated S-Function block mask does not display these parameters.

Run-Time Parameters and S-Function Compatibility Diagnostics

If you set the **S-function upgrades needed** option on the **Diagnostics > Compatibility** pane of the Configuration Parameters dialog box to **warning** or **error**, the code generator instructs you to upgrade S-functions that you create with the **Generate S-function** feature. This is because the S-function system target file does

not register run-time parameters. Run-time parameters are only supported for inlined S-Functions and the generated S-Function supports features that prevent it from being inlined (for example, it can call or contain other noninlined S-functions).

You can work around this limitation by setting the **S-function upgrades needed** option to none.

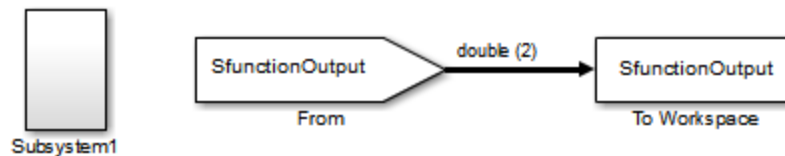
Limitations on Using Goto and From Block

When using the S-function system target file, the code generator restricts I/O to correspond to the root model Inport and Outport blocks (or the Inport and Outport blocks of the Subsystem block from which the S-function target was generated). No code is generated for Goto or From blocks.

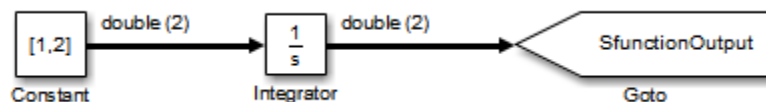
To work around this restriction, create your model and subsystem with the required Inport and Outport blocks, instead of using Goto and From blocks to pass data between the root model and subsystem. In the model that incorporates the generated S-function, you would then add Goto and From blocks.

Example Before Work Around

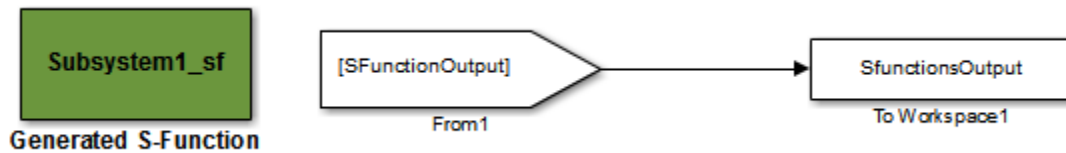
- Root model with a From block and subsystem, `Subsystem1`



- `Subsystem1` with a Goto block, which has global visibility and passes its input to the `From` block in the root model

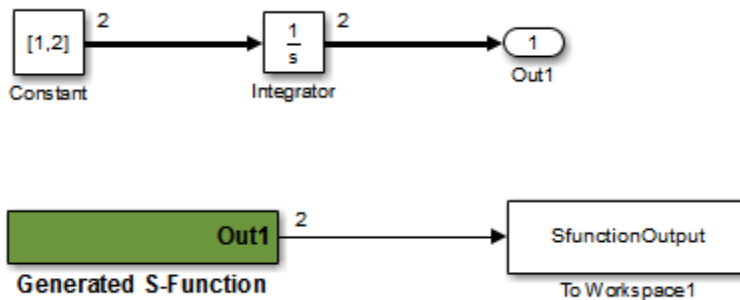


- `Subsystem1` replaced with an S-function generated with the S-Function target — a warning results when you run the model because the generated S-function does not implement the Goto block



Example After Work Around

An Outport block replaces the GoTo block in `Subsystem1`. When you plug the generated S-function into the root model, its output connects directly to the To Workspace block.



Limitations on Building and Updating S-Functions

The following limitations apply to building and updating S-functions using the S-function system target file:

- You cannot build models that contain Model blocks using the S-function system target file. This also means that you cannot build a subsystem module by right-clicking (or by using **Code > C/C++ Code > Build Selected Subsystem**) if the subsystem contains Model blocks. This restriction applies only to S-functions generated using the S-function target, not to ERT S-functions.
- If you modify the model that generated an S-Function block, the build process does not automatically rebuild models containing the generated S-Function block. This is in contrast to the practice of automatically rebuilding models referenced by Model blocks when they are modified (depending on the Model Reference **Rebuild** configuration setting).

- Handwritten S-functions without corresponding TLC files must contain exception-free code. For more information on exception-free code, see “Exception Free Code” in the Simulink documentation.

Unsupported Blocks

The S-function format does not support the following built-in blocks:

- Interpreted MATLAB Function block
- S-Function blocks containing any of the following:
 - MATLAB language S-functions (unless you supply a TLC file for C code generation)
 - Fortran S-functions (unless you supply a TLC file for C code generation)
 - C/C++ MEX S-functions that call into the MATLAB environment
- Scope block
- To Workspace block

The S-function format does not support blocks from the `embeddedtargetslib` block library.

SimState Not Supported for Code Generation

You can use `SimState` within C-MEX and Level-2 MATLAB language S-functions to save and restore the simulation state (see “S-Function Compliance with the `SimState`” in the Simulink documentation). However, `SimState` is not supported for code generation, including with the S-function system target file.

Profiling Code Performance Not Supported

Profiling the performance of generated code using the Target Language Compiler (TLC) hook function interface described by the example model `rtwdemo_profile` is not supported for the S-function target.

Limitations on Nesting S-Functions

The following limitations apply to nesting a generated S-Function block in a model or subsystem from which you generate another S-function:

- The software does not support nonvirtual bus input and output signals for a nested S-function.

- You should avoid nesting an S-function in a model or subsystem having the same name as the S-function (possibly several levels apart). In such situations, the S-function can be called recursively. The software currently does not detect such loops in S-function dependency, which can result in aborting or hanging your MATLAB session. To prevent this from happening, be sure to name the subsystem or model to be generated as an S-function target uniquely, to avoid duplicating existing MEX filenames on the MATLAB path.

Limitations on User-Defined Data Types

The S-function system target file does not support the `HeaderFile` property that can be specified on user-defined data types, including those based on `Simulink.AliasType`, `Simulink.Bus`, and `Simulink.NumericType` objects. If a user-defined data type in your model uses the `HeaderFile` property to specify an associated header file, code generation with the S-function system target file disregards the value and does not generate a corresponding include statement.

Limitation on Right-Click Generation of an S-Function Target

If you generate an S-function target by right-clicking a Function-Call Subsystem block, the original subsystem and the generated S-function might not be consistent. An inconsistency occurs when the **States when enabling** parameter of the Trigger Port block inside the Function-Call Subsystem block is set to **inherit**. You must set the **States when enabling** parameter to **reset** or **held**, otherwise Simulink reports an error.

Limitation on S-Functions with Bus I/O Signals

If an S-function generated using the S-function target has bus input or output signals, the generated bus data structures might include padding to align fields of the bus elements with the Simulink representation used during simulation. However, if you insert the S-function in a model and generate code using a model target such as `grt.tlc`, the bus structure alignment generated for the model build might be incompatible with the padding generated for the S-function and might affect the numerical results of code execution. To make the structure alignment consistent between model simulation and execution of the model code, for each `Simulink.Bus` object, you can modify the `HeaderFile` property to remove the unpadded bus structure header file. This will cause the bus typedefs generated for the S-function to be reused in the model code.

Limitation on Subsystems with Function-Call I/O Signals

The S-function target does not support creating an S-Function block from a subsystem that has a function-call trigger input or a function-call output.

Data Store Access

When an S-Function in your model accesses a data store during simulation, Simulink disables data store diagnostics.

- If you created the S-Function from a model, the diagnostic is disabled for global data stores as well.
- If you created the S-Function from a subsystem, the diagnostic is disabled for the following data stores:
 - Global data stores
 - Data stores placed outside the subsystem, but accessed by Data Store Read or Data Store Write blocks.

Cannot Specify Output Dimensions Through Subsystem Mask

You cannot specify **Port dimensions** for an Outport block through a subsystem mask if you want to generate an S-Function block from the subsystem. The software produces an error when you try to run a simulation that uses the S-Function block, for example:

```
Invalid setting in 'testSystem/Subsystem/__OutputSSForSFun__/Out2'  
for parameter 'PortDimensions'  
...
```

More About

- “Acceleration”
- “Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” on page 22-2

Real-Time Systems in Simulink Coder

- “Deploy Algorithm Model for Real-Time Rapid Prototyping” on page 23-2
- “Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation” on page 23-5

Deploy Algorithm Model for Real-Time Rapid Prototyping

Use the code generator to deploy algorithm models for real-time rapid prototyping.

In this section...

“About Real-Time Rapid Prototyping” on page 23-2

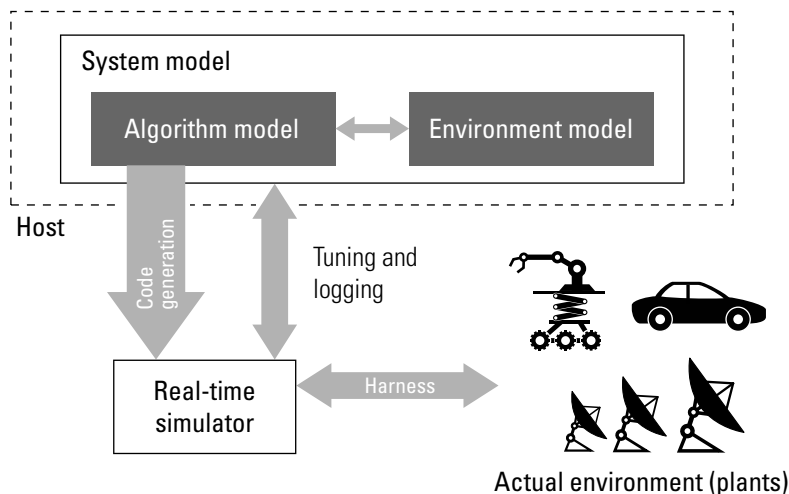
“Goals of Real-Time Rapid Prototyping” on page 23-2

“Refine Code With Real-Time Rapid Prototyping” on page 23-3

About Real-Time Rapid Prototyping

Real-time rapid prototyping requires the use of a real-time simulator, potentially connected to system hardware (for example, physical plant or vehicle) being controlled. You generate, deploy, and tune code as it runs on the real-time simulator or embedded microprocessor. This design step is crucial for verifying whether a component can adequately control the system, and allows you to assess, interact with, and optimize code.

The following figure shows a typical approach for real-time rapid prototyping.



Goals of Real-Time Rapid Prototyping

Assuming that you have documented functional requirements, refined concept models, system hardware for the physical plant or vehicle being controlled, and access to target

products you intend to use (for example, for example, the Simulink Real-Time or Simulink Desktop Real-Time product), you can use real-time prototyping to:

- Refine component and environment model designs by rapidly iterating between algorithm design and prototyping
- Validate whether a component can adequately control the physical system in real time
- Evaluate system performance before laying out hardware, coding production software, or committing to a fixed design
- Test hardware

Refine Code With Real-Time Rapid Prototyping

To perform real-time rapid prototyping:

- 1 Create or acquire a real-time system that runs in real time on rapid prototyping hardware. The Simulink Real-Time product facilitates real-time rapid prototyping. This product provides a real-time operating system that makes PCs run in real time. It also provides device driver blocks for numerous hardware I/O cards. You can then create a rapid prototyping system using inexpensive commercial-off-the-shelf (COTS) hardware. In addition, third-party vendors offer products based on the Simulink Real-Time product or other code generation technology that you can integrate into a development environment.
- 2 Use provided system target files to generate code that you can deploy onto a real-time simulator. See the following information.

Engineering Tasks	Related Product Information	Examples
Generate code for real-time rapid prototyping	<p>“System Target Files and Code Generation Features” on page 17-18</p> <p>“Event-Based Scheduling”</p> <p>— Embedded Coder</p> <p>“Support for Standards and Guidelines” — Embedded Coder</p>	rtwdemo_counter

Engineering Tasks	Related Product Information	Examples
Generate code for rapid prototyping in hard real time, using PCs	Simulink Real-Time “Code Generation Configuration Parameters” in the Simulink Real-Time documentation	help xpcdemos
Generate code for rapid prototyping in soft real time, using PCs	Simulink Desktop Real-Time “Code Generation Configuration Parameters” in the Simulink Desktop Real-Time documentation	sldrtex_vdp (and others)

3 Monitor signals, tune parameters, and log data.

More About

- “Access Signal, State, and Parameter Data During Execution” on page 14-3
- “Rapid Control Prototyping Process”

Deploy Environment Model for Real-Time Hardware-In-the-Loop (HIL) Simulation

In this section...

“About Hardware-In-the-Loop Simulation” on page 23-5

“Set Up and Run HIL Simulations” on page 23-6

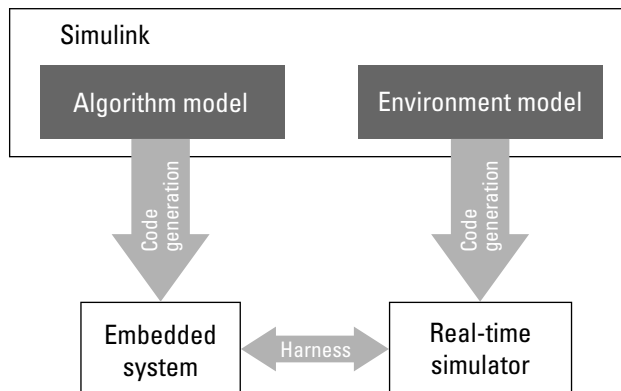
About Hardware-In-the-Loop Simulation

Hardware-in-the-loop (HIL) simulation tests and verifies an embedded system or control unit in the context of a software test platform. Examples of test platforms include real-time target systems and instruction set simulators (IISs). You use Simulink software to develop and verify a model that represents the test environment. Using the code generator, you produce, build, and download an executable program for the model to the HIL simulation platform. After you set up the environment, you can run the executable to validate the embedded system or control unit in real time.

During HIL simulation, you gradually replace parts of a system environment with hardware components as you refine and fabricate the components. HIL simulation offers an efficient design process that eliminates costly iterations of part fabrication.

The code that you build for the system simulator provides real-time system capabilities. For example, the code can include VxWorks from Wind River or another real-time operating system (RTOS).

The following figure shows a typical HIL setup.



The HIL platform available from MathWorks is the Simulink Real-Time product. Several third-party products are also available for use as HIL platforms. The Simulink Real-Time product offers hard real-time performance for PCs with Intel or AMD® 32-bit processors functioning as your real-time target. The Simulink Real-Time product enables you to add I/O interface blocks to your models and automatically generate code with code generation technology. The Simulink Real-Time product can download the code to a second PC running the Simulink Real-Time real-time kernel. System integrator solutions that are based on Simulink Real-Time are also available.

Set Up and Run HIL Simulations

To set up and run HIL simulations iterate through the following steps:

- 1** Develop a model that represents the environment or system under development. For more information, see:
 - “System Target Files and Code Generation Features” on page 17-18
- 2** Generate an executable for the environment model.
- 3** Download the executable for the environment model to the HIL simulation platform.
- 4** Replace software representing a system component with corresponding hardware.
- 5** Test the hardware in the context of the HIL system.
- 6** Repeat steps 4 and 5 until you can simulate the system after including components that require testing.

More About

- “Access Signal, State, and Parameter Data During Execution” on page 14-3
- “Hardware-in-the-Loop Simulation Process”

External Code Integration in Simulink Coder

- “About External Code Integration” on page 24-2
- “Choose External Code Integration Approach” on page 24-3
- “Automate S-Function Generation with S-Function Builder” on page 24-22
- “Build S-Function With Legacy Code Tool” on page 24-27
- “Use C++ Object Methods and Objects with Legacy Code Tool” on page 24-31
- “External Code Integration Examples” on page 24-37
- “Configure Model for External Code Integration” on page 24-83
- “Insert Code into Model Custom Code Blocks” on page 24-86
- “Insert External Code into Stateflow Charts” on page 24-94
- “S-Functions and Code Generation” on page 24-99
- “Write S-Function and TLC Files By Hand” on page 24-104
- “Build Support for S-Functions” on page 24-160

About External Code Integration

The code generator includes a variety of approaches for integrating legacy or custom code with generated code. *Legacy code* is existing handwritten code or code for environments that you need to integrate with code produced by the code generator. *Custom code* is legacy code or other user-specified lines of code that you need to include in the code generator build process. Collectively, legacy and custom code are called *external code*.

You can achieve external code integration in two ways. You can import existing external code into code produced by code generation technology or you can export generated code into an existing external code base. For example, you might want to use generated code as a plug-in function.

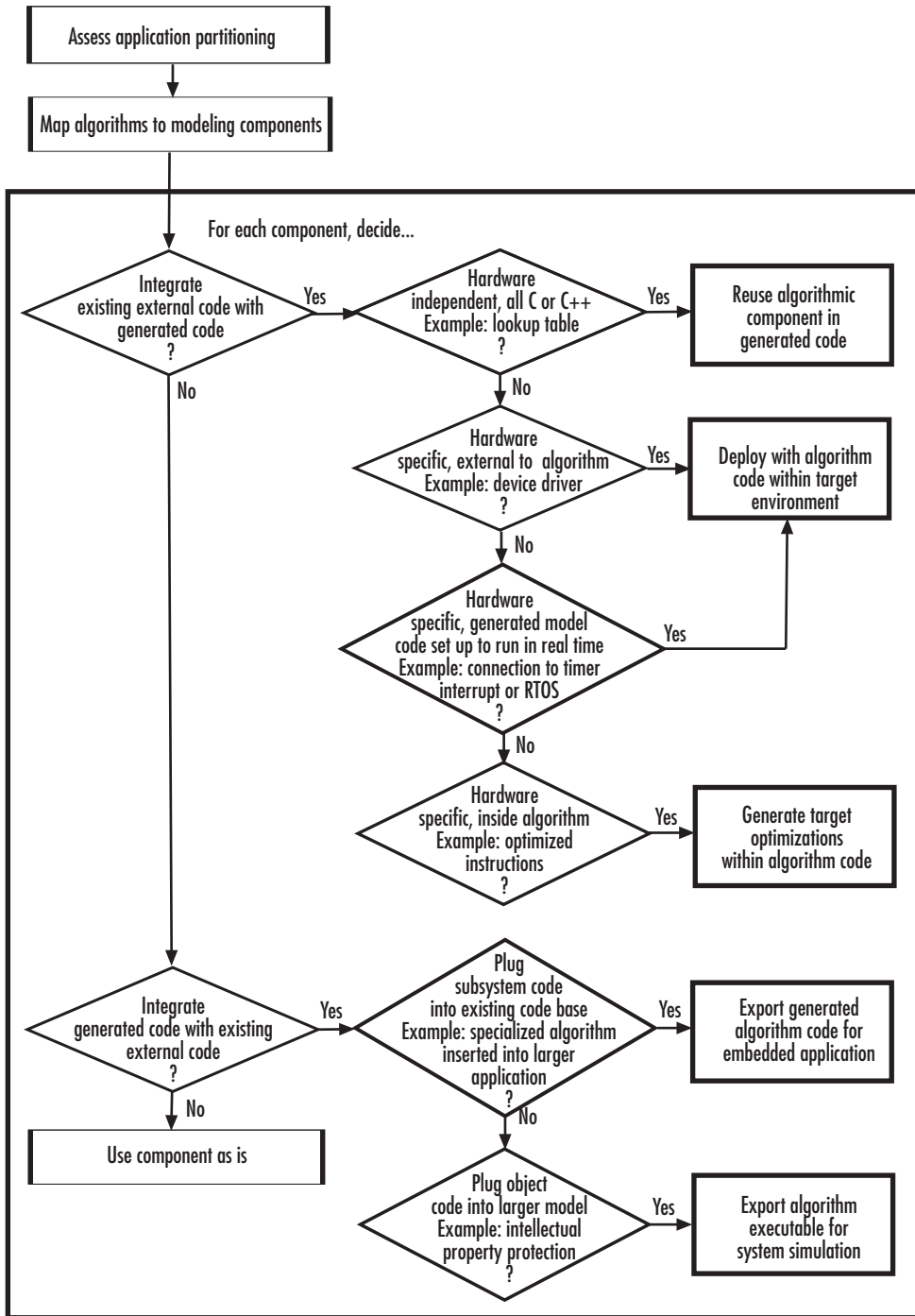
See “Choose External Code Integration Approach” on page 24-3 for guidance on choosing an approach based on your application.

Choose External Code Integration Approach

Based on application goals, external code integration can be characterized as follows:

- Import external code into generated code
 - Reuse of an algorithmic component in generated code
 - Deploy an application with algorithm code within the target environment
 - Generate target optimizations within algorithm code
- Export generated code into external code
 - Export generated algorithm code for an embedded application
 - Export an algorithm executable for system simulation

Use the following flow diagram to prepare for integration and choose integration paths that best map to your application components. As the diagram shows, before you make integration decisions, assess your application architecture and partition it as much as possible. Working with smaller units makes it easier to map algorithms to modeling components and decide how to integrate the components. For each component, use the highlighted area of the flow diagram to identify the most applicable type of integration. Then, see the information provided for the corresponding type.



In this section...

- “Reuse Algorithmic Components in Generated Code” on page 24-5
- “Deploy Algorithm Code Within a Target Environment” on page 24-13
- “Export Generated Algorithm Code for Embedded Applications” on page 24-18
- “Export Algorithm Executables for System Simulation” on page 24-20
- “Modify External Code for Language Compatibility” on page 24-20

Reuse Algorithmic Components in Generated Code

The approach you choose to reuse algorithmic components in generated code depends on the language used to program the components.

- “Reusable Algorithmic Components” on page 24-5
- “Integrate External MATLAB Code” on page 24-6
- “Integrate External C or C++ Code” on page 24-8
- “Integrate Fortran Code” on page 24-12
- “Integration Considerations for Reusable Algorithmic Components” on page 24-12

Reusable Algorithmic Components

You have several options for integrating reusable algorithmic components with the generated code. Such components are hardware-independent and can be verified with Simulink simulation. Examples of such components include:

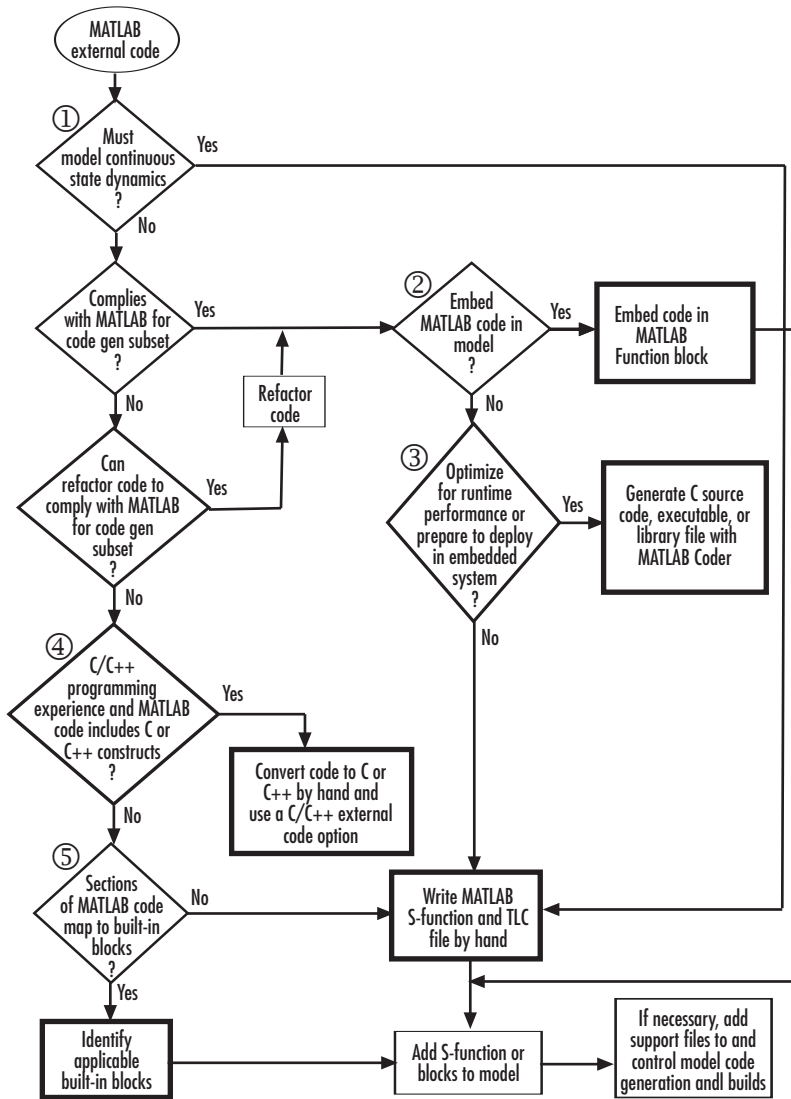
- Lookup tables
- Math utilities
- Digital filters
- Special integrators
- Proportional–integral–derivative (PID) control modules

Some integration options allow you to integrate external code for a reusable component directly, while other options convert external code to modeling elements. To take full advantage of Model-Based Design, convert code to modeling elements which you can then use in the Simulink or Stateflow simulation environment. Doing so enables you to simulate and generate code for an integrated component and, for example, use software-

in-the-loop (SIL) or processor-in-the-loop (PIL) testing to verify whether algorithm behavior is the same in both environments.

Integrate External MATLAB Code

The following diagram identifies common characteristics or requirements for reusable algorithmic components written in MATLAB code and recommends solutions in each case. The table that follows the diagram provides more detail. Collectively, the diagram and table help you choose the best solution for your application and find more related information.



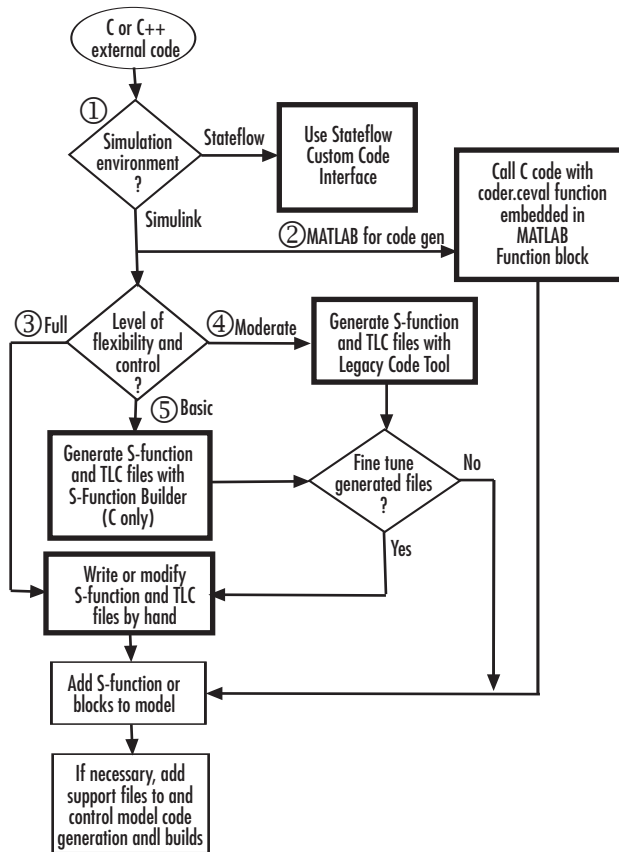
	If...	Then...	For More Information, See...
1	The algorithm must model continuous state dynamics	Write a MATLAB S-function and, for generating code, a corresponding TLC	<ul style="list-style-type: none"> • “MATLAB S-Functions” • “Target Language Compiler”

	If...	Then...	For More Information, See...
		file for the algorithm and add the S-function to your model	
2	You want to embed MATLAB code directly in the model	Add a MATLAB Function block to the model and embed the MATLAB code in that block	MATLAB Function
3	You need to optimize runtime performance or prepare to deploy the code in an embedded system	Use MATLAB Coder software to generate a C source code, executable, or library file	“Getting Started with MATLAB Coder”
4	You have C or C++ programming experience and the external MATLAB code is compact and primarily uses C or C++ constructs	Convert the MATLAB code to C or C++ code manually and choose an option for integrating the C or C++ code	“Integrate External C or C++ Code” on page 24-8
5	Sections of the external MATLAB code map to built-in blocks	Develop the algorithm in the context of a model, using the applicable built in blocks	<ul style="list-style-type: none"> • “Model Editing Fundamentals” and “Component-Based Modeling” • “Supported Products and Block Usage” on page 2-4

To embed external MATLAB code in a MATLAB Function block or generate C or C++ code from MATLAB code with the MATLAB Coder software, the MATLAB code must use functions suitable for code generation.

Integrate External C or C++ Code

The following diagram identifies common characteristics or requirements for reusable algorithmic components written in C or C++ code and recommends solutions in each case. The table that follows the diagram provides more detail. Collectively, the diagram and table will help you choose the best solution for your application and find more related information.



	If...	Then...	For More Information, See...
1	The external code will simulate in a Stateflow environment	Use the Stateflow custom code interface	<ul style="list-style-type: none"> sf_custom “Call Custom C Code Functions” in the Stateflow documentation
2	Performance is not an issue and you want to quickly embed a call to external C or C++ code in a model	Call the C or C++ code with the <code>coder.ceval</code> function from a MATLAB Function block	<ul style="list-style-type: none"> <code>coder.ceval</code> function description MATLAB Function block
3	You want maximum flexibility and the	Write an S-function and TLC file manually	<ul style="list-style-type: none"> “C S-Function Examples” and “C++ S-Function Examples”

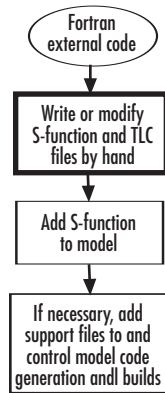
	If...	Then...	For More Information, See...
	ability to control what code is generated; the application requires function overloading or you need to format data definitions such that they are compatible with a function		<ul style="list-style-type: none"> • “S-Function Basics” • “S-Functions and Code Generation” on page 24-99
4	You want ease of use with moderate flexibility to control what code gets generated, typically for discrete applications; you have C or C++ programming experience, but prefer to generate the files that add the code to a model; optimizing generated code is essential	Use the Legacy Code Tool to generate the S-function and TLC files; optionally, you can fine-tune the generated files manually to better meet application needs	<ul style="list-style-type: none"> • “Integrate External C Functions That Implement N-Dimensional Table Lookups” • “Integrate C Functions Using Legacy Code Tool” in the Simulink documentation • “Build S-Function With Legacy Code Tool” on page 24-27
5	You want ease of use with basic flexibility to control what code gets generated, typically for mixed discrete and continuous-time applications; programming experience is limited or the external code requires a Fixed-Point block interface	Use the S-Function Builder to generate the S-function and TLC files; optionally, you can fine tune the generated files manually to better meet application needs	“Build S-Functions Automatically” in the Simulink documentation

If you must control how code generation technology declares, stores, and represents data in generated code, you can do so by designing (creating) and applying custom storage classes (CSCs) if you have an Embedded Coder license. For information on CSCs, see

the examples `rtwdemo_cscpredef`, `rtwdemo_importstruct`, `rtwdemo_advsc`, and “Custom Storage Classes” in the Embedded Coder documentation.

Integrate Fortran Code

The following diagram shows that to integrate external Fortran code as reusable algorithmic components you must integrate the code by writing an S-function and corresponding TLC file.



For information, see “Fortran S-Function Examples” and “Fortran S-Functions”.

Integration Considerations for Reusable Algorithmic Components

Note: Solutions marked with *EC only* require an Embedded Coder license.

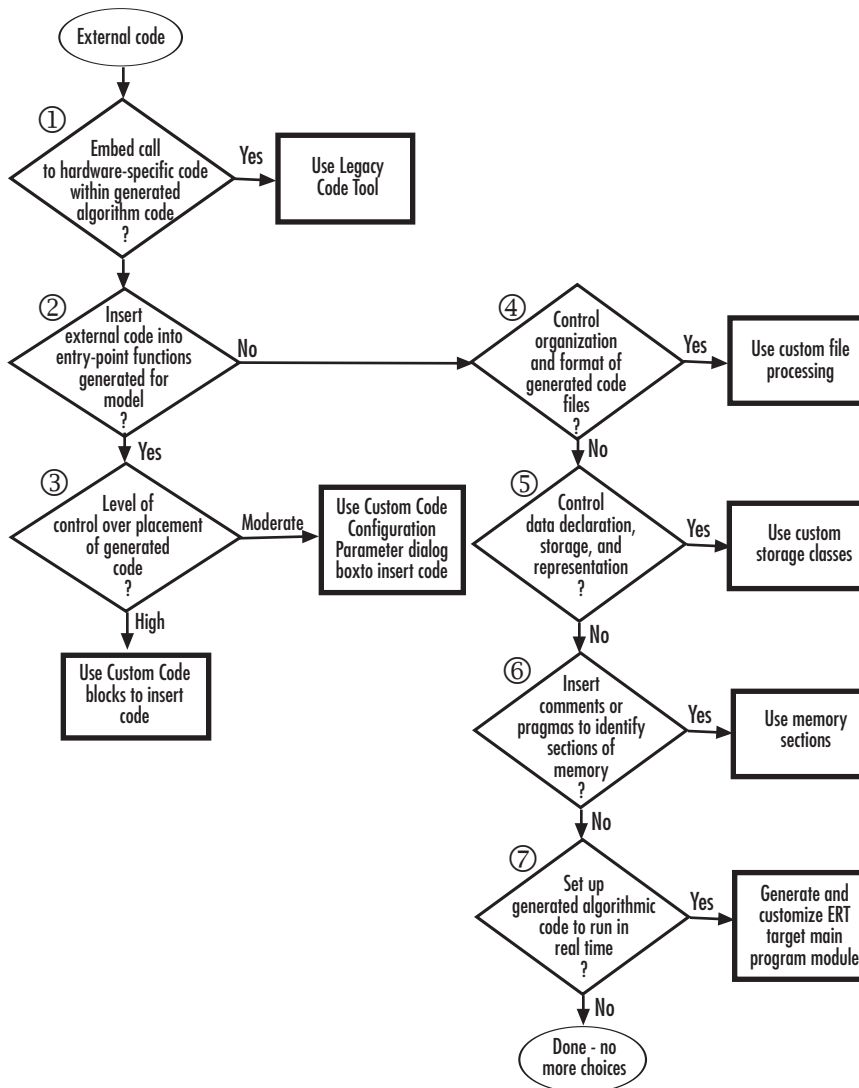
If...	Consider...	For More Information, See...
Generated code must use math functions and operators that are consistent with imported external code	<i>EC only</i> —Using the Code Replacement Tool, code replacement library (CRL) API, and Code Replacement Viewer to create, examine, validate, and register function and operator code replacement tables	<ul style="list-style-type: none"> “Code Replacement Customization” “Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®”
Generated code must share data with imported external code	Using data creation and management technologies, such as Simulink data objects, alias and numeric data types,	<ul style="list-style-type: none"> “Data Representation” “Exchange and Reuse Parameter Data Between

If...	Consider...	For More Information, See...
	and data type replacement to match data type, formatting, and storage that is consistent with that used by the imported external code	Generated Code and Existing Code” <ul style="list-style-type: none"> • “Introduction to Custom Storage Classes” • “Configure Generated Code According to Interface Control Document”
Style and format of identifiers in generated code must be consistent with style and format applied in imported external code	In the Configuration Parameters dialog box, on the Symbols pane, configure identifier naming for the generated code	<ul style="list-style-type: none"> • <code>rtwdemo_symbols</code> • <code>rtwdemo_namerules</code> • “Construction of Generated Identifiers” on page 16-14 and “Customize Generated Identifier Naming Rules” in the Embedded Coder documentation
Use of comments in generated code must match the use of comments in imported external code	In the Configuration Parameters dialog box, on the Comments pane, configure comments for the generated code	<ul style="list-style-type: none"> • <code>rtwdemo_comments</code> • “Configure Code Comments” on page 16-13 and “Add Custom Comments to Generated Code” in the Embedded Coder documentation
Style of code, such as style and usage of parentheses, must be match the style used in imported external code	<i>EC only</i> —In the Configuration Parameters dialog box, on the Code Style pane, configure the style for the generated code	<ul style="list-style-type: none"> • <code>rtwdemo_parentheses</code> • “Control Code Style” in the Embedded Coder documentation

Deploy Algorithm Code Within a Target Environment

Code generation technology can generate a single set of application source files from an algorithm model and integrated external C or C++ code that supports the target environment hardware. For example, you might have a working device driver that you want to integrate with algorithmic code that has to read data from and write data to the I/O device the driver supports. Typically, a deployed model algorithm calls out to the external code.

The following diagram identifies common characteristics or requirements for a target environment in which generated algorithm code might be deployed and recommends solutions. The table that follows the diagram provides more detail. Collectively, the diagram and table help you choose the best solution for your application and find more related information.



Note: Solutions marked with *EC only* require an Embedded Coder license.

	If You Need To...	Then...	For More Information, See...
1	Embed a call to hardware-specific code, such as a device driver, within generated algorithm code	Use the Legacy Code Tool	<ul style="list-style-type: none"> • “Device Drivers” on page 37-116 • “Integrate C Functions Using Legacy Code Tool” in the Simulink documentation • “Build S-Function With Legacy Code Tool” on page 24-27
2	Insert target-specific C or C++ code into entry-point functions that the code generator produces for a model with a high level of control over code placement; for example, inserting startup, initialization, or termination code	Use Custom Code blocks	<ul style="list-style-type: none"> • <code>rtwdemo_slcustcode</code> • “Insert Code into Model Custom Code Blocks” on page 24-86
3	Insert application-specific C or C++ code—near the top of the generated source code or header file or inside the model initialization or termination function—or files for the build process—source, header, library—for the external code	Configure files and data for the external code environment by entering code and file specifications for parameters on the Code Generation > Interface pane of the Configuration Parameters dialog box	<ul style="list-style-type: none"> • “Integrate Generated Code with Custom Code in External Environment” • “Configure Model for External Code Integration” on page 24-83

	If You Need To...	Then...	For More Information, See...
4	Control the organization and format of code files generated for a model—for example, placement of code in sections, inclusion of banners, calls to generated entry-point functions, generation of a main program module	<i>EC only</i> —Use the custom file processing components—code generation template (CGT) files, code template API, and custom file processing (CFP) templates (the API and CFP templates require TLC programming knowledge)	<ul style="list-style-type: none"> • <code>rtwdemo_codetemplate</code> • “Customize Code Organization and Format” in the Embedded Coder documentation
5	Control how the code generator declares, stores, and represents signals, tunable parameters, block states, and data objects in generated code	<i>EC only</i> — Design (create) and apply custom storage classes	<ul style="list-style-type: none"> • “Introduction to Custom Storage Classes” • “Control Signals and States in Code by Applying Storage Classes” on page 14-120 • “Block Parameter Representation in the Generated Code” on page 14-43
6	Insert comments or pragmas in generated code to identify memory for custom storage classes or model- or subsystem-level functions and internal data	<i>EC only</i> — Use the memory section capability	<ul style="list-style-type: none"> • <code>rtwdemo_memsec</code> • “Control Data and Function Placement in Memory by Inserting Pragmas”
7	Set up generated algorithmic code to run in real time—within the context of a real-time operating system (RTOS) or on hardware that is not running an operating system (bare board)	<i>EC only</i> —Generate and customize an ERT target main (harness) program module (<code>ert_main.c</code> or <code>ert_main.cpp</code>) for the model	<ul style="list-style-type: none"> • “Deployment” • “Deploy Generated Standalone Executables To Target Hardware”

Export Generated Algorithm Code for Embedded Applications

You have multiple options for configuring and preparing a model or subsystem so that you can plug its generated source code into an existing external code base.

Scan the first column of the following table to identify tasks that apply to the algorithm code you want to export. For each task that applies, the information in the corresponding row describes how to achieve the goal, using code generation technology.

Note: Solutions marked with *EC only* require an Embedded Coder license.

If You Need To...	Then...	For More Information, See...
Insert C or C++ code into specific entry-point functions that the code generator produces for interfacing with the external code	Use Custom Code blocks	<ul style="list-style-type: none"> • <code>rtwdemo_slcustcode</code> • “Insert Code into Model Custom Code Blocks” on page 24-86
Pass composite data	Represent the data in the model as a vector or structure	<ul style="list-style-type: none"> • <code>rtwdemo_scalarrep</code> • <code>rtwdemo_slbus</code> • “Group Signals into Structures in the Generated Code Using Buses” on page 14-136 • “Use <code>memcpy</code> Function to Optimize Generated Code for Vector Assignments” on page 31-56
Read from or write to a specific region or area of memory	<i>EC only</i> — Set up a Data Store Memory block in the model to represent the area of memory and define the area with the built-in advanced custom storage class (CSC) <code>GetSet</code>	“Data Stores in Generated Code” on page 14-31
Generate a C++ class interface to the model code — encapsulating	Select <code>C++ class</code> code interface packaging in the Configuration	<ul style="list-style-type: none"> • “Simple Use of C++ Class Control”

If You Need To...	Then...	For More Information, See...
model data as properties and model entry-point functions as methods	Parameters dialog box or programmatically with MATLAB commands — customizing the generated C++ class interface is <i>EC only</i>	<ul style="list-style-type: none"> • “Control Generation of C++ Class Interfaces” in the Embedded Coder documentation
Control how the code generator produces function prototypes — arguments, argument order, and data types—for a model (for example, so the prototypes match the external code)	<i>EC only</i> — Configure function prototypes for the model by clicking the Configure Model Functions button on the Code Generation > Interface pane of the Configuration Parameters dialog box and entering data in the Model Interface dialog box; alternatively, configure the function prototypes programmatically in the MATLAB command window or with a script	<ul style="list-style-type: none"> • “Sample Procedure for Configuring Function Prototypes” • “Control Generation of Function Prototypes” in the Embedded Coder documentation
Insert application-specific C or C++ code —near the top of the generated source code or header file or inside the model initialization or termination function—or files for the build process —source, header, library—for the external code	Configure files and data for the external code environment by entering code and file specifications for parameters on the Code Generation > Interface pane of the Configuration Parameters dialog box	<ul style="list-style-type: none"> • “Integrate Generated Code with Custom Code in External Environment” • “Configure Model for External Code Integration” on page 24-83

If You Need To...	Then...	For More Information, See...
Generate code, which is to be integrated with an existing C code base, for a function-call or virtual subsystem	<i>EC only</i> — Review and adjust for exported subsystem requirements, configure the parent model to use an ERT target, and right-click build the subsystem block using the C/C++ Code > Export Functions menu item	<ul style="list-style-type: none"> • “Techniques for Exporting Function-Call Subsystems” • <code>rtwdemo_exporting_functions</code> • “Export Function-Call Subsystems” in the Embedded Coder documentation

Export Algorithm Executables for System Simulation

If you have an Embedded Coder license, you can use an ERT shared library target (`ert_shrlib.tlc`) to build a Windows dynamic link library (`.dll`), a UNIX shared object (`.so`) file, or a Macintosh OS X dynamic library (`.dylib`) file from a model. An application, which runs on a Windows, UNIX, or Macintosh OS X system, can then load the shared library file. You can upgrade a shared library without having to recompile applications that use it.

Uses of shared library files include:

- Adding a software component to an application for system simulation
- Reusing off-the-shelf software modules among applications on a host system
- Hiding source code (intellectual property) for software shared with a vendor

For an example, see `rtwdemo_shrlib`. For more information, see “Package Generated Code as Shared Object Libraries” in the Embedded Coder documentation.

Modify External Code for Language Compatibility

If you need to integrate external C code with generated C++ code or vice versa, you must modify your external code to be language compatible with the generated code. Options for making the code language compatible include:

- Writing or rewriting the legacy or custom code in the same language as the generated code.
- If the generated code is in C++ and your legacy or custom code is in C, for each C function, create a header file that prototypes the function, using the following format:

```
#ifdef __cplusplus
extern "C" {
#endif
int my_c_function_wrapper();
#ifdef __cplusplus
}
#endif
```

The prototype serves as a function wrapper. If your compiler supports C++ code, the value `__cplusplus` is defined. The linkage specification `extern "C"` specifies C linkage with no name mangling.

- If the generated code is in C and your legacy or custom code is in C++, include an `extern "C"` linkage specification in each `.cpp` file. For example, the following shows a portion of C++ code in the file `my_func.cpp`:

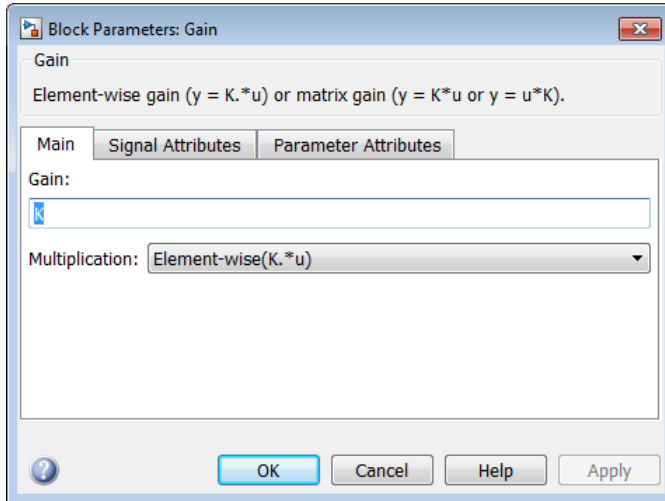
```
extern "C" {

int my_cpp_function()
{
    ...
}
}
```

Automate S-Function Generation with S-Function Builder

The **Generate S-function** feature automates the process of generating an S-function from a subsystem. In addition, the **Generate S-function** feature presents a display of parameters used within the subsystem, and lets you declare selected parameters tunable.

As an example, consider `SourceSubsys`, the same subsystem illustrated in the example “Create S-Function Blocks from a Subsystem” on page 22-36. The objective is to automatically extract `SourceSubsys` from the model and build an S-Function block from it, as in the previous example. In addition, the workspace variable `K`, which is the gain factor of the Gain block within `SourceSubsys` (as shown in the Gain block parameter dialog box below), is declared and generated as a tunable variable.



To auto-generate an S-function from `SourceSubsys` with tunable parameter `K`,

- 1 With the `SourceSubsys` model open, click the subsystem to select it.
- 2 From the **Code** menu, select **C/C++ Code > Generate S-Function**. This menu item is enabled when a subsystem is selected in the current model.

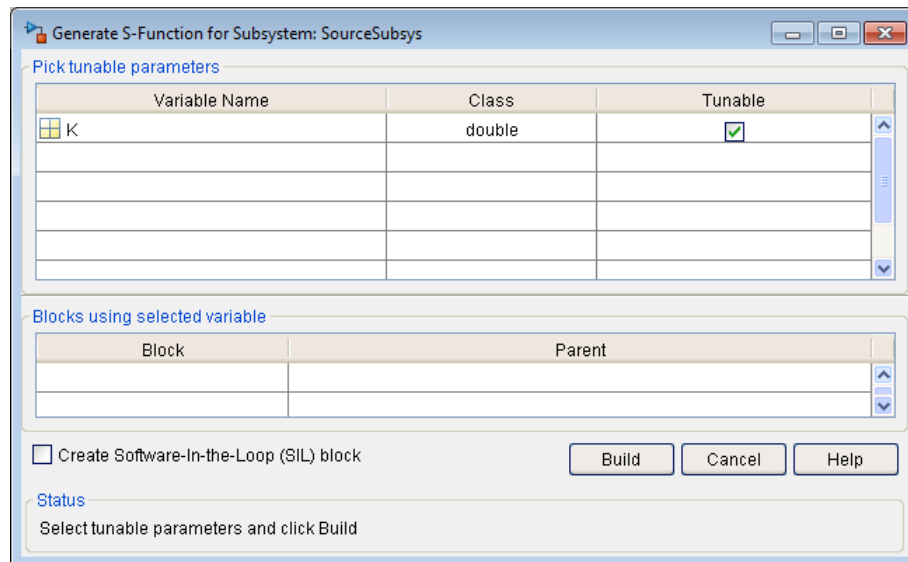
Alternatively, you can right-click the subsystem and select **C/C++ Code > Generate S-Function** from the subsystem block's context menu.

- 3 The **Generate S-Function** window is displayed (see the next figure). This window shows variables (or data objects) that are referenced as block parameters in the subsystem, and lets you declare them as tunable.

The upper pane of the window displays three columns:

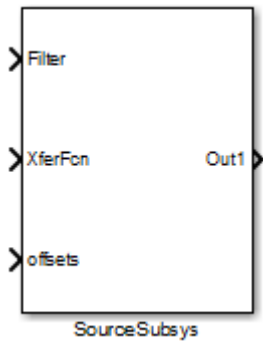
- **Variable Name:** name of the parameter.
- **Class:** If the parameter is a workspace variable, its data type is shown. If the parameter is a data object, its name and class is shown
- **Tunable:** Lets you select tunable parameters. To declare a parameter tunable, select the check box. In the next figure, the parameter K is declared tunable.

When you select a parameter in the upper pane, the lower pane shows the blocks that reference the parameter, and the parent system of each such block.



Generate S-Function Window

- 4 After selecting tunable parameters, click the **Build** button. This initiates code generation and compilation of the S-function, using the S-function target. The **Create New Model** option is automatically enabled.
- 5 The build process displays status messages in the MATLAB Command Window. When the build completes, the tunable parameters window closes, and a new untitled model window opens.



- 6 The model window contains an S-Function block with the same name as the subsystem from which the block was generated (in this example, **SourceSubsys**). Optionally, you can save the generated model containing the generated block.
- 7 The generated code for the S-Function block is stored in the current working folder. The following files are written to the top level folder:
 - *subsys_sf.c* or *.cpp*, where *subsys* is the subsystem name (for example, *SourceSubsys_sf.c*)
 - *subsys_sf.h*
 - *subsys_sf.mexext*, where *mexext* is a platform-dependent MEX-file extension (for example, *SourceSubsys_sf.mexw64*)

The source code for the S-function is written to the subfolder *subsys_sfcn_rtw*. The top-level *.c* or *.cpp* file is a stub file that simply contains an include directive that you can use to interface other C/C++ code to the generated code.

Note: For a list of files required to deploy your S-Function block for simulation or code generation, see “Required Files for S-Function Deployment” on page 22-34.

- 8 The generated S-Function block has inports and outports whose widths and sample times correspond to those of the original model.

The following code, from the `mdlOutputs` routine of the generated S-function code (in *SourceSubsys_sf.c*), shows how the tunable variable `K` is referenced by using calls to the MEX API.

```
static void mdlOutputs(SimStruct *S, int_T tid)
```



```

...
/* Gain: '<S1>/Gain' incorporates:
 * Sum: '<S1>/Sum'
 */
rtb_Gain_n[0] = (rtb_Product_p + (((const
real_T**)ssGetInputPortSignalPtrs(S, 2))[0]))) * ((real_T
*)(mxGetData(K(S))));
rtb_Gain_n[1] = (rtb_Product_p + (((const
real_T**)ssGetInputPortSignalPtrs(S, 2))[1]))) * ((real_T
*)(mxGetData(K(S))));

```

- In automatic S-function generation, the **Use Value for Tunable Parameters** option is cleared or at the command line is set to 'off'.
- Use a MEX S-function wrapper only in the MATLAB version in which the wrapper is created.

If you specify paths and files with absolute

Macro Parameters

Suppose that you apply a custom storage class such as `Define` to a `Simulink.Parameter` object so that the parameter appears as a macro in the generated code. If you use the parameter object inside a subsystem from which you generate an ERT S-function, you cannot select the parameter object as a tunable parameter. Instead, the S-function code generator applies the custom storage class to the parameter object. This generation of macros in the S-function code allows you to generate S-functions from subsystems that contain variant elements, such as Variant Subsystem blocks, that you configure to produce preprocessor conditionals in the generated code. However, you cannot change the value of the parameter during simulation of the S-function.

To select the parameter object as a tunable parameter, apply a different storage class or custom storage class. Custom storage classes that treat parameters as macros include `Define`, `ImportedDefine`, `CompilerFlag`, and custom storage classes that you create by setting **Data initialization** to **Macro** in the Custom Storage Class Designer. If you use a non-macro storage class or custom storage class, you cannot use the parameter object as a variant control variable and generate preprocessor conditionals.

If you apply a custom storage class that treats the parameter object as an imported macro, provide the macro definition before you generate the ERT S-function. For example, suppose you apply the custom storage class `ImportedDefine` to a `Simulink.Parameter` object, and use the parameter object as a variant control variable in the subsystem. If you set the custom attribute `HeaderFile` to 'myHdr.h', when you

generate the S-function, place the custom header file `myHdr.h` in the current folder. The generated S-function uses the macro value from your header file instead of the value from the `Value` property of the parameter object.

To use a macro that you define through a compiler option, for example by applying the custom storage class `CompilerFlag`, use the model configuration parameter **Configuration Parameters > Code Generation > Custom Code > Additional build information > Defines** to specify the compiler option. For more information, see [Code Generation Pane: Custom Code: Additional Build Information: Defines](#).

See Also

`legacy_code`

More About

- “S-Functions and Code Generation” on page 24-99
- “Build S-Function With Legacy Code Tool” on page 24-27

Build S-Function With Legacy Code Tool

Legacy Code Tool and Code Generation

You can use the Simulink Legacy Code Tool to automatically generate fully inlined C MEX S-functions for legacy or custom code that is optimized for embedded components, such as device drivers and lookup tables, that call existing C or C++ functions.

Note: The Legacy Code Tool can interface with C++ functions, but not C++ objects. For a work around so that the tool can interface with C++ objects, see “Legacy Code Tool Limitations” in the Simulink documentation.

You can use the tool to:

- Compile and build the generated S-function for simulation.
- Generate a masked S-Function block that is configured to call the existing external code.

If you want to include these types of S-functions in models for which you intend to generate code, you must use the tool to generate a TLC block file. The TLC block file specifies how the generated code for a model calls the existing C or C++ function.

If the S-function depends on files in folders other than the folder containing the S-function dynamically loadable executable file, and you want to maintain those dependencies for building a model that includes the S-function, use the tool to also generate an `rtwmakecfg.m` file for the S-function. For example, for some applications, such as custom targets, you might want to locate files in a target-specific location. The build process looks for the generated file `rtwmakecfg.m` in the same folder as the S-function dynamically loadable executable and calls the `rtwmakecfg` function if the software finds the file.

For more information, see “Integrate C Functions Using Legacy Code Tool” in the Simulink documentation.

Generate Inlined S-Function Files for Code Generation

Depending on your application's code generation requirements, to generate code for a model that uses the S-function, you can choose to do either of the following:

- Generate one `.cpp` file for the inlined S-function. In the Legacy Code Tool data structure, set the value of the `Options.singleCPPMexFile` field to `true` before generating the S-function source file from your existing C function. For example:

```
def.Options.singleCPPMexFile = true;
legacy_code('sfcn_cmex_generate', def);
```

- Generate a source file and a TLC block file for the inlined S-function. For example:

```
def.Options.singleCPPMexFile = false;
legacy_code('sfcn_cmex_generate', def);
legacy_code('sfcn_tlc_generate', def);
```

singleCPPMexFile Limitations

You cannot set the `singleCPPMexFile` field to `true` if

- `Options.language='C++'`
- You use one of the following Simulink objects with the `IsAlias` property set to `true`:
 - `Simulink.Bus`
 - `Simulink.AliasType`
 - `Simulink.NumericType`
- The Legacy Code Tool function specification includes a `void*` or `void**` to represent scalar work data for a state argument
- `HeaderFiles` field of the Legacy Code Tool structure specifies multiple header files

Apply Code Style Settings to Legacy Functions

To apply the model configuration parameters for code style to a legacy function:

- 1 Initialize the Legacy Code Tool data structure. For example:

```
def = legacy_code('initialize');
```

- 2 In the data structure, set the value of the `Options.singleCPPMexFile` field to `true`. For example:

```
def.Options.singleCPPMexFile = true;
```

To verify the setting, enter:

```
def.Options.singleCPPMexFile
```

singleCPPMexFile Limitations

You cannot set the `singleCPPMexFile` field to `true` if

- `Options.language='C++'`
- You use one of the following Simulink objects with the `IsAlias` property set to `true`:
 - `Simulink.Bus`
 - `Simulink.AliasType`
 - `Simulink.NumericType`
- The Legacy Code Tool function specification includes a `void*` or `void**` to represent scalar work data for a state argument
- `HeaderFiles` field of the Legacy Code Tool structure specifies multiple header files

Address Dependencies on Files in Different Locations

By default, the Legacy Code Tool assumes that files on which an S-function depends reside in the same folder as the dynamically loadable executable file for the S-function. If your S-function depends on files that reside elsewhere and you are using the template makefile build process, you must generate an `rtwmakecfg.m` file for the S-function. For example, it is likely that you need to generate this file if your Legacy Code Tool data structure defines compilation resources as path names.

To generate the `rtwmakecfg.m` file, call the `legacy_code` function with `'rtwmakecfg_generate'` as the first argument, and the name of the Legacy Code Tool data structure as the second argument.

```
legacy_code('rtwmakecfg_generate', lct_spec);
```

If you use multiple registration files in the same folder and generate an S-function for each file with a single call to `legacy_code`, the call to `legacy_code` that specifies `'rtwmakecfg_generate'` must be common to all registration files. For more information, see “Handling Multiple Registration Files” in the Simulink documentation

For example, if you define `defs` as an array of Legacy Code Tool structures, you call `legacy_code` with `'rtwmakecfg_generate'` once.

```
defs = [defs1(:);defs2(:);defs3(:)];
legacy_code('rtwmakecfg_generate', defs);
```

For more information, see “Build Support for S-Functions” on page 24-160.

Deploy S-Functions for Simulation and Code Generation

You can deploy the S-functions that you generate with the Legacy Code Tool so that other people can use them. To deploy an S-function for simulation and code generation, share the following files:

- Registration file
- Compiled dynamically loadable executable
- TLC block file
- `rtwmakecfg.m` file
- Header, source, and include files on which the generated S-function depends

Users of the deployed files must be aware that:

- Before using the deployed files in a Simulink model, they must add the folder that contains the S-function files to the MATLAB path.
- If the Legacy Code Tool data structure registers required files as absolute paths and the location of the files changes, they must regenerate the `rtwmakecfg.m` file.

Use C++ Object Methods and Objects with Legacy Code Tool

Integrate External C++ Object Methods

Integrate legacy C++ object methods by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C++ MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a block TLC file and optional rtwmakecfg.m file that calls the legacy code during code generation.

Provide the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The definition of the legacy C++ class in this example is:

```
class adder {
private:
    int int_state;
public:
    adder();
    int add_one(int increment);
    int get_val();
};
```

The legacy source code is in the files `adder_cpp.h` and `adder_cpp.cpp`.

```
% rtwdemo_sfun_adder_cpp
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_adder_cpp';
def.StartFcnSpec = 'createAdder()';
def.OutputFcnSpec = 'int32 y1 = adderOutput(int32 u1)';
def.TerminateFcnSpec = 'deleteAdder()';
```

```
def.HeaderFiles = {'adder_cpp.h'};
def.SourceFiles = {'adder_cpp.cpp'};
def.IncPaths    = {'rtwdemo_lct_src'};
def.SrcPaths    = {'rtwdemo_lct_src'};
def.Options.Language = 'C++';
def.Options.useTlcWithAccel = false;
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function `legacy_code()` again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file `rtwdemo_sfun_adder_cpp.cpp`.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function `legacy_code()` again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_adder_cpp
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_1');
Building with 'Microsoft Visual C++ 2013 Professional'.
MEX completed successfully.
    mex('rtwdemo_sfun_adder_cpp.cpp', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src');
Building with 'Microsoft Visual C++ 2013 Professional'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_adder_cpp
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function `legacy_code()` again. Set the first input to 'sfcn_tlc_generate' to generate a TLC block file that supports code generation through Simulink® Coder™. If the TLC block file is not created and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: `rtwdemo_sfun_adder_cpp.tlc`.

```
legacy_code('sfcn_tlc_generate', def);
```


Generate an `rtwmakecfg.m` File for Code Generation

After you create the TLC block file, you can call the function `legacy_code()` again. Set the first input to `'rtwmakecfg_generate'` to generate an `rtwmakecfg.m` file that supports code generation through Simulink® Coder™. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the `rtwmakecfg.m` file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function `legacy_code()` again. Set the first input to `'slblock_generate'` to generate a masked S-function block that is configured to call that S-function. The software places the block in a new model. You can copy the block to an existing model.

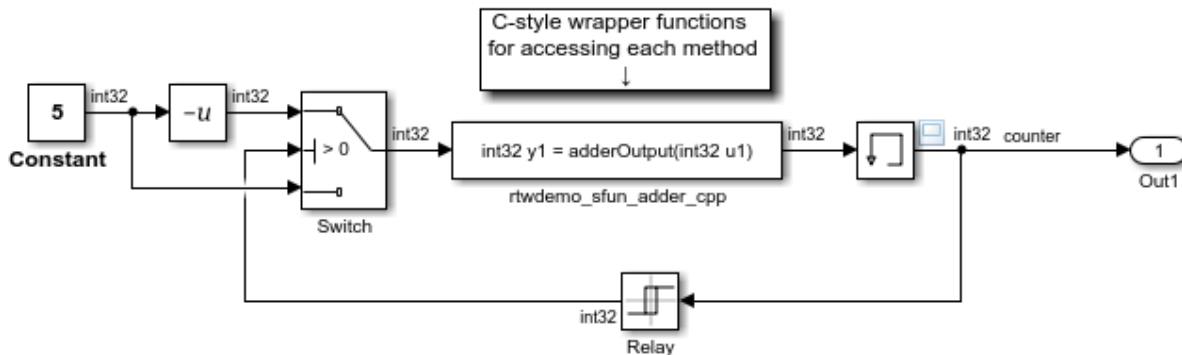
```
% legacy_code('slblock_generate', def);
```

Show the Generated Integration with Legacy Code

The model `rtwdemo_lct_cpp` shows integration with the legacy code.

```
open_system('rtwdemo_lct_cpp')  
sim('rtwdemo_lct_cpp')
```

This example illustrates how the Legacy Code Tool is used to call C++ object member functions



For this example to run you must go to the MATLAB command prompt and enter: `>> mex -setup` then select a C++ compiler.

To view the sources for this example click the links below.

- open legacy files `adder_cpp.h` and `adder_cpp.cpp`
- open registration script `rtwdemo_lct_cpp_script.m`
- open generated S-Function `rtwdemo_sfun_adder_cpp.cpp`
- open generated TLC file `rtwdemo_sfun_adder_cpp.tlc`

Generate Code Using
Embedded Coder
(double-click)

Copyright 1990-2012 The MathWorks, Inc.

Integrate External C++ Objects

The Legacy Code Tool can interface with C++ functions, but not C++ objects. Using the previous example as a starting point, here is an example of how you can work around this limitation.

- Modify the class definition for `adder` in a new file `adder_cpp.hpp`. Add three new macros that dynamically allocate a new `adder` object, invoke the method `add_one()`, and free the memory allocated. Each macro takes a pointer to an `adder` object. Because each function called by the Legacy Code Tool must have a C-like signature,

the pointer is cached and passed as a `void*`. Then you must explicitly cast to `adder*` in the macro. The new class definition for `adder`:

```
#ifndef _ADDER_CPP_
#define _ADDER_CPP_

class adder {
private:
    int int_state;
public:
    adder(): int_state(0) {};
    int add_one(int increment);
    int get_val() {return int_state;};
};

// Method wrappers implemented as macros
#define createAdder(work1) \
    *(work1) = new adder

#define deleteAdder(work1) \
    delete(static_cast<adder*>(*(work1)))

#define adderOutput(work1, u1) \
    (static_cast<adder*> (*(work1)))->add_one(u1)

#endif /* _ADDER_CPP_ */
```

- Update `adder_cpp.cpp`. With the class modification, instead of one global instance, each generated S-function manages its own `adder` object.

```
#include "adder_cpp.hpp"

int adder::add_one(int increment)
{
    int_state += increment;
    return int_state;
}
```

- Update `rtwdemo_sfun_adder_cpp.cpp` with the following changes:
 - `StartFcnSpec` calls the macro that allocates a new `adder` object and caches the pointer.

```
def.StartFcnSpec = 'createAdder(void **work1)';
```

- `OutputFcnSpec` calls the macro that invokes the method `add_one()` and provides the S-function specific `adder` pointer object.

```
def.OutputFcnSpec = 'int32 y1 = adderOutput(void *work1, int32 u1)';
```

- `TerminateFcnSpec` calls the macro that frees the memory.

```
def.TerminateFcnSpec = 'deleteAdder(void **work1)';
```

See Also

`legacy_code`

Related Examples

- “S-Functions and Code Generation” on page 24-99
- “Build S-Function With Legacy Code Tool” on page 24-27

External Code Integration Examples

This topic shows various scenarios of external code integration.

In this section...

“Integrate External C Functions That Pass Input Arguments By Value Versus Address” on page 24-38

“Integrate External C Functions That Pass the Output Argument As a Return Argument” on page 24-42

“Integrate External C Functions That Pass Input and Output Arguments as Signals with a Fixed-Point Data Type” on page 24-45

“Integrate External C Functions with Instance-Specific Persistent Memory” on page 24-48

“Integrate External C Functions That Use Structure Arguments” on page 24-51

“Integrate External C Functions That Pass Input and Output Arguments as Signals with Complex Data” on page 24-54

“Integrate External C Functions That Pass Arguments That Have Inherited Dimensions” on page 24-57

“Integrate External C Functions That Implement Start and Terminate Actions” on page 24-61

“Integrate External C Functions That Pass Arguments as Multi-Dimensional Signals” on page 24-64

“Integrate External C Functions with a Block Sample Time Specified, Inherited, and Parameterized” on page 24-67

“Insert External C and C++ Code Into Stateflow Charts for Code Generation” on page 24-72

“Integrate External C Code Into Generated Code Using Custom Code Blocks and Model Configuration Parameters” on page 24-74

“Integrate External C Code Into Generated Code Using Custom Code Blocks and Model Configuration Parameters” on page 24-77

“Insert External C and C++ Code Into Stateflow Charts for Code Generation” on page 24-79

Integrate External C Functions That Pass Input Arguments By Value Versus Address

This example shows how to use the Legacy Code Tool to integrate legacy C functions that pass their input arguments by value versus address.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional `rtwmakecfg.m` file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototypes of the legacy functions being called in this example are:

- `FLT filterV1(const FLT signal, const FLT prevSignal, const FLT gain)`
- `FLT filterV2(const FLT* signal, const FLT prevSignal, const FLT gain)`

FLT is a typedef to float. The legacy source code is in the files `your_types.h`, `myfilter.h`, `filterV1.c`, and `filterV2.c`.

Note the difference in the `OutputFcnSpec` defined in the two structures; the first case specifies that the first input argument is passed by value, while the second case specifies pass by pointer.

```
defs = [];  
  
% rtwdemo_sfun_filterV1  
def = legacy_code('initialize');  
def.SFunctionName = 'rtwdemo_sfun_filterV1';  
def.OutputFcnSpec = 'single y1 = filterV1(single u1, single u2, single p1)';
```

```

def.HeaderFiles = {'myfilter.h'};
def.SourceFiles = {'filterV1.c'};
def.IncPaths    = {'rtwdemo_lct_src'};
def.SrcPaths    = {'rtwdemo_lct_src'};
defs = [defs; def];

% rtwdemo_sfun_filterV2
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_filterV2';
def.OutputFcnSpec = 'single y1 = filterV2(single u1[1], single u2, single p1)';
def.HeaderFiles   = {'myfilter.h'};
def.SourceFiles   = {'filterV2.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];

```

Generate S-Functions for Simulation

To generate C-MEX S-functions according to the description provided by the input argument 'defs', call the function `legacy_code()` again with the first input set to 'sfcn_cmex_generate'. The S-functions call the legacy functions in simulation. The source code for the S-functions is in the files `rtwdemo_sfun_filterV1.c` and `rtwdemo_sfun_filterV2.c`.

```
legacy_code('sfcn_cmex_generate', defs);
```

Compile the Generated S-Functions for Simulation

After you generate the C-MEX S-function source files, to compile the S-functions for simulation with Simulink®, call the function `legacy_code()` again with the first input set to 'compile'.

```
legacy_code('compile', defs);
```

```

### Start Compiling rtwdemo_sfun_filterV1
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_')
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_filterV1.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src',
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_filterV1
### Exit

```

```
### Start Compiling rtwdemo_sfun_filterV2
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_filterV2.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src',
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_filterV2
### Exit
```

Generate TLC Block Files for Code Generation

After you compile the S-functions and use them in simulation, you can call the function `legacy_code()` again with the first input set to `'sfcn_tlc_generate'` to generate TLC block files. Block files specify how the generated code for a model calls the legacy code. If you do not generate TLC block files and you try to generate code for a model that includes the S-functions, code generation fails. The TLC block files for the S-functions are `rtwdemo_sfun_filterV1.tlc` and `rtwdemo_sfun_filterV2.tlc`.

```
legacy_code('sfcn_tlc_generate', defs);
```

Generate an `rtwmakecfg.m` File for Code Generation

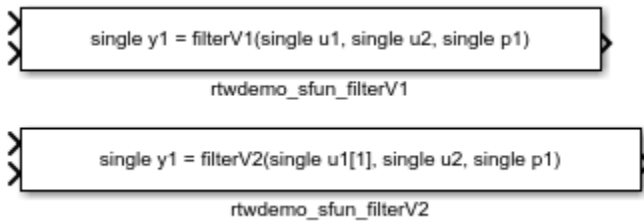
After you create the TLC block files, you can call the function `legacy_code()` again with the first input set to `'rtwmakecfg_generate'` to generate an `rtwmakecfg.m` file to support code generation. If the required source and header files for the S-functions are not in the same folder as the S-functions, and you want to add these dependencies in the makefile produced during code generation, generate the `rtwmakecfg.m` file.

```
legacy_code('rtwmakecfg_generate', defs);
```

Generate Masked S-Function Blocks for Calling the Generated S-Functions

After you compile the C-MEX S-function source, you can call the function `legacy_code()` again with the first input set to `'slblock_generate'` to generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. From there you can copy them to an existing model.

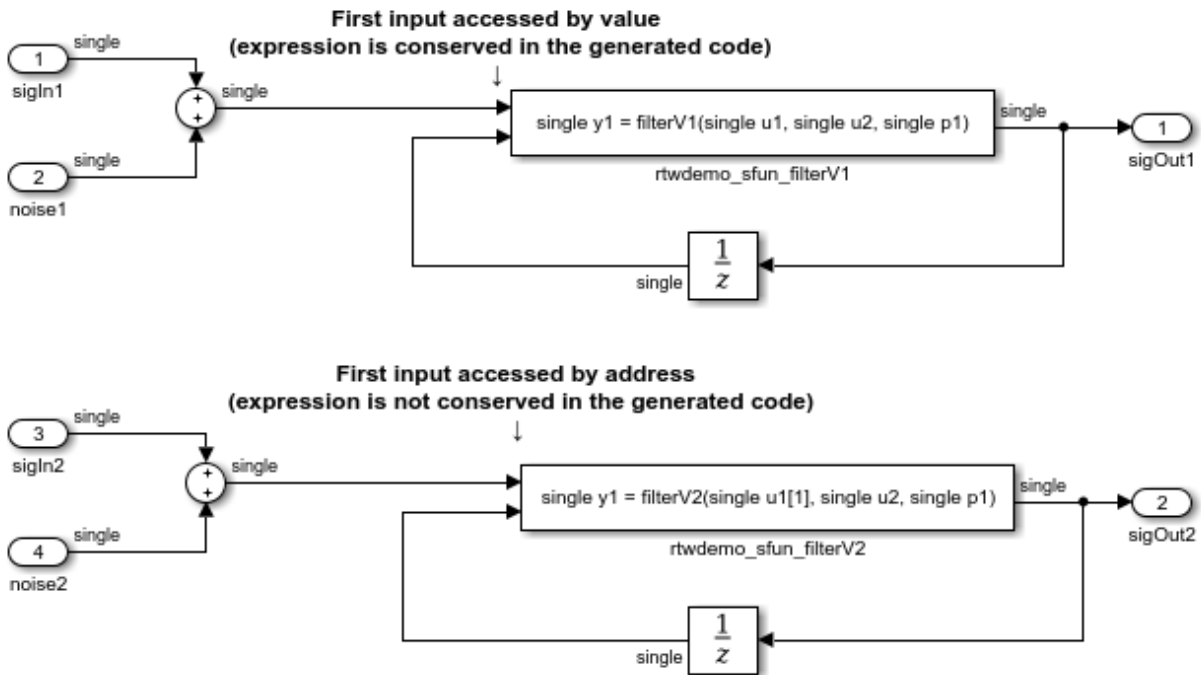
```
legacy_code('slblock_generate', defs);
```

Show the Generated Integration with Legacy Code

The model `rtwdemo_lct_filter` shows integration of the model with the legacy code. The subsystem `TestFilter` serves as a harness for the calls to the legacy C functions via the generate S-functions, with unit delays serving to store the previous output values.

```
open_system('rtwdemo_lct_filter')
open_system('rtwdemo_lct_filter/TestFilter')
sim('rtwdemo_lct_filter')
```



Integrate External C Functions That Pass the Output Argument As a Return Argument

This example shows how to use the Legacy Code Tool to integrate legacy C functions that pass their output as a return argument.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional `rtwmakecfg.m` file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototype of the legacy functions being called in this example is:

```
FLT gainScalar(const FLT in, const FLT gain)
```

FLT is a typedef to float. The legacy source code is in the files `your_types.h`, `gain.h`, and `gainScalar.c`.

```
% rtwdemo_sfun_gain_scalar
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_gain_scalar';
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';
def.HeaderFiles    = {'gain.h'};
def.SourceFiles    = {'gainScalar.c'};
def.IncPaths       = {'rtwdemo_lct_src'};
def.SrcPaths       = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function `legacy_code()` again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file `rtwdemo_sfun_gain_scalar.c`.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function `legacy_code()` again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_gain_scalar
mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_
```

```

Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfund_gain_scalar.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfund_gain_scalar
### Exit

```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function `legacy_code()` again with the first input set to `'sfcn_tlc_generate'` to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: `rtwdemo_sfund_gain_scalar.tlc`.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an `rtwmakecfg.m` File for Code Generation

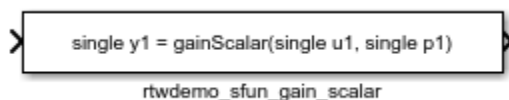
After you create the TLC block file, you can call the function `legacy_code()` again with the first input set to `'rtwmakecfg_generate'` to generate an `rtwmakecfg.m` file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the `rtwmakecfg.m` file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function `legacy_code()` again with the first input set to `'slblock_generate'` to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

```
legacy_code('slblock_generate', def);
```



Show the Generated Integration with Legacy Code

The model `rtwdemo_lct_gain` shows integration of the model with the legacy code. The subsystem `TestGain` serves as a harness for the call to the legacy C function via the generate S-function.

```
open_system('rtwdemo_lct_gain')
open_system('rtwdemo_lct_gain/TestGain')
sim('rtwdemo_lct_gain')
```



Integrate External C Functions That Pass Input and Output Arguments as Signals with a Fixed-Point Data Type

This example shows how to use the Legacy Code Tool to integrate legacy C functions that pass their inputs and outputs by using parameters of fixed-point data type.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional `rtwmakecfg.m` file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototype of the legacy functions being called in this example is:

```
myFixpt timesS16(const myFixpt in1, const myFixpt in2, const uint8_T fracLength)
```

myFixpt is logically a fixed-point data type, which is physically a typedef to a 16-bit integer:

```
myFixpt = Simulink.NumericType;
myFixpt.DataTypeMode = 'Fixed-point: binary point scaling';
myFixpt.Signed = true;
myFixpt.WordLength = 16;
myFixpt.FractionLength = 10;
myFixpt.IsAlias = true;
myFixpt.HeaderFile = 'timesFixpt.h';
```

The legacy source code is in the files timesFixpt.h, and timesS16.c.

```
% rtwdemo_sfun_times_s16
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_times_s16';
def.OutputFcnSpec = 'myFixpt y1 = timesS16(myFixpt u1, myFixpt u2, uint8 p1)';
def.HeaderFiles = {'timesFixpt.h'};
def.SourceFiles = {'timesS16.c'};
def.IncPaths = {'rtwdemo_lct_src'};
def.SrcPaths = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_times_s16.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_times_s16
mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_
```

```

Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfundtimes_s16.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfundtimes_s16
### Exit

```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function `legacy_code()` again with the first input set to `'sfcn_tlc_generate'` to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: `rtwdemo_sfundtimes_s16.tlc`.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an `rtwmakecfg.m` File for Code Generation

After you create the TLC block file, you can call the function `legacy_code()` again with the first input set to `'rtwmakecfg_generate'` to generate an `rtwmakecfg.m` file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the `rtwmakecfg.m` file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function `legacy_code()` again with the first input set to `'slblock_generate'` to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

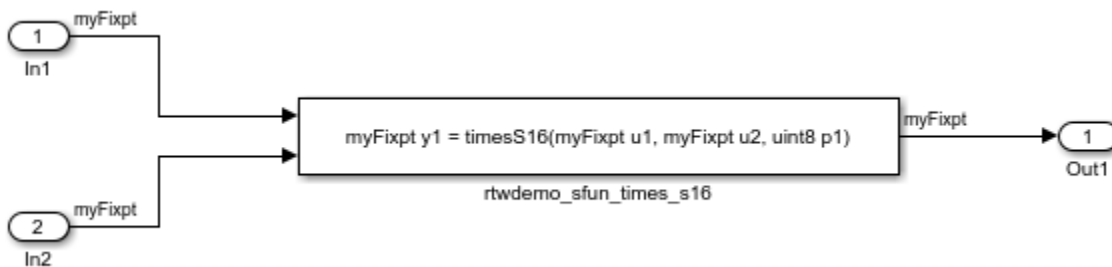
```
legacy_code('slblock_generate', def);
```



Show the Integration of the Model with Legacy Code

The model `rtwdemo_lct_fixpt_signals` shows integration of the model with the legacy code. The subsystem `TestFixpt` serves as a harness for the call to the legacy C function via the generated S-function.

```
open_system('rtwdemo_lct_fixpt_signals')
open_system('rtwdemo_lct_fixpt_signals/TestFixpt')
sim('rtwdemo_lct_fixpt_signals')
```



Integrate External C Functions with Instance-Specific Persistent Memory

Integrate legacy C functions that use instance-specific persistent memory by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional `rtwmakecfg.m` file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the

properties, call `legacy_code('help')`. The prototypes of the legacy functions being called in this example are:

```
void memory_bus_init(COUNTERBUS *mem, int32_T upper_sat, int32_T lower_sat);
```

```
void memory_bus_step(COUNTERBUS *input, COUNTERBUS *mem, COUNTERBUS *output);
```

`mem` is an instance-specific persistent memory for applying a one integration step delay. `COUNTERBUS` is a struct typedef defined in `counterbus.h` and implemented with a Simulink.Bus object in the base workspace. The legacy source code is in the files `memory_bus.h`, and `memory_bus.c`.

```
evalin('base', 'load rtwdemo_lct_data.mat')

% rtwdemo_sfunk_work
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfunk_work';
def.InitializeConditionsFcnSpec = ...
    'void memory_bus_init(COUNTERBUS work1[1], int32 p1, int32 p2)';
def.OutputFcnSpec = ...
    'void memory_bus_step(COUNTERBUS u1[1], COUNTERBUS work1[1], COUNTERBUS y1[1])';
def.HeaderFiles    = {'memory_bus.h'};
def.SourceFiles    = {'memory_bus.c'};
def.IncPaths       = {'rtwdemo_lct_src'};
def.SrcPaths       = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument `def`, call the function `legacy_code()` again with the first input set to `'sfcn_cmex_generate'`. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file `rtwdemo_sfunk_work.c`.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function `legacy_code()` again with the first input set to `'compile'`.

```
legacy_code('compile', def);
```

```

### Start Compiling rtwdemo_sfun_work
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_work.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-I
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_work
### Exit

```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function `legacy_code()` again with the first input set to `'sfcn_tlc_generate'` to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: `rtwdemo_sfun_work.tlc`.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an `rtwmakecfg.m` File for Code Generation

After you create the TLC block file, you can call the function `legacy_code()` again with the first input set to `'rtwmakecfg_generate'` to generate an `rtwmakecfg.m` file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the `rtwmakecfg.m` file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function `legacy_code()` again with the first input set to `'slblock_generate'` to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

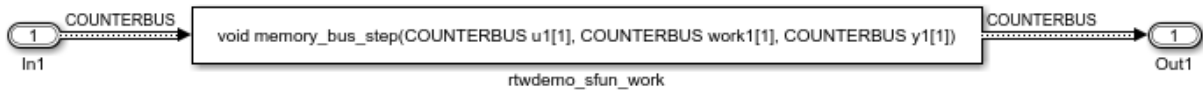
```
legacy_code('slblock_generate', def);
```



Show the Integration of the Model with Legacy Code

The model `rtwdemo_lct_work` shows integration of the model with the legacy code. The subsystem `memory_bus` serves as a harness for the call to the legacy C function.

```
open_system('rtwdemo_lct_work')
open_system('rtwdemo_lct_work/memory_bus')
sim('rtwdemo_lct_work')
```



This legacy function apply a one integration step delay. The output is the previous input value. The parameters P1 and P2 set the initial values of the sub structure fields "upper_saturation_limit" and "lower_saturation_limit"

Integrate External C Functions That Use Structure Arguments

Integrate legacy C functions with structure arguments that use Simulink® buses with the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional `rtwmakecfg.m` file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototype of the legacy functions being called in this example is:

```
counterbusFcn(COUNTERBUS *u1, int32_T u2, COUNTERBUS *y1, int32_T *y2)
```

COUNTERBUS is a struct typedef defined in counterbus.h and implemented with a Simulink.Bus object in the base workspace. The legacy source code is in the files counterbus.h, and counterbus.c.

```
evalin('base','load rtwdemo_lct_data.mat')

% rtwdemo_sfun_counterbus
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_counterbus';
def.OutputFcnSpec = ...
    'void counterbusFcn(COUNTERBUS u1[1], int32 u2, COUNTERBUS y1[1], int32 y2[1])';
def.HeaderFiles = {'counterbus.h'};
def.SourceFiles = {'counterbus.c'};
def.IncPaths = {'rtwdemo_lct_src'};
def.SrcPaths = {'rtwdemo_lct_src'};
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function legacy_code() again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file rtwdemo_sfun_counterbus.c.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function legacy_code() again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_counterbus
mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_1');
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
mex('rtwdemo_sfun_counterbus.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src');
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_counterbus
```

```
### Exit
```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function `legacy_code()` again with the first input set to `'sfcn_tlc_generate'` to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: `rtwdemo_sfun_counterbus.tlc`.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an `rtwmakecfg.m` File for Code Generation

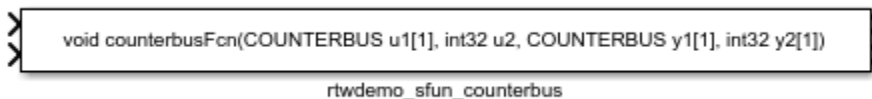
After you create the TLC block file, you can call the function `legacy_code()` again with the first input set to `'rtwmakecfg_generate'` to generate an `rtwmakecfg.m` file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the `rtwmakecfg.m` file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function `legacy_code()` again with the first input set to `'slblock_generate'` to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

```
legacy_code('slblock_generate', def);
```



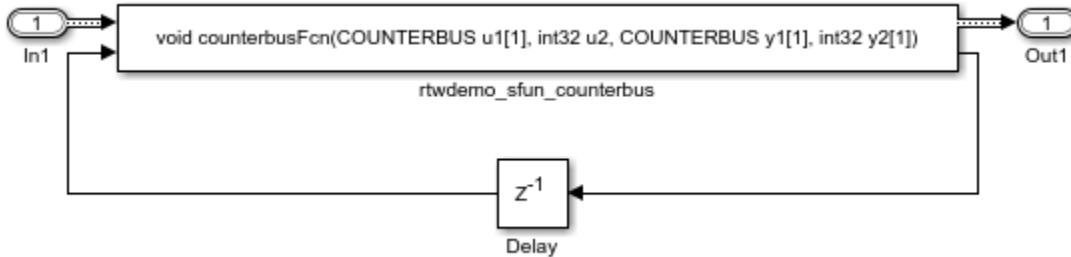
```
void counterbusFcn(COUNTERBUS u1[1], int32 u2, COUNTERBUS y1[1], int32 y2[1])
rtwdemo_sfun_counterbus
```

Show the Integration of the Model with Legacy Code

The model `rtwdemo_lct_bus` shows integration of the model with the legacy code. The subsystem `TestCounter` serves as a harness for the call to the legacy C function.

```
open_system('rtwdemo_lct_bus')
```

```
open_system('rtwdemo_lct_bus/TestCounter')
sim('rtwdemo_lct_bus')
```



Integrate External C Functions That Pass Input and Output Arguments as Signals with Complex Data

Integrate legacy C functions using complex signals with the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional `rtwmakecfg.m` file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototype of the legacy functions being called in this example is:

```
void cplx_gain(creal_T *input, creal_T *gain, creal_T *output);
```

`creal_T` is the complex representation of a double. The legacy source code is in the files `cplxgain.h`, and `cplxgain.c`.

```

% rtwdemo_sfun_gain_scalar
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_cplx_gain';
def.OutputFcnSpec = ...
    'void cplx_gain(complex<double> u1[1], complex<double> p1[1], complex<double> y1[1]';
def.HeaderFiles = {'cplxgain.h'};
def.SourceFiles = {'cplxgain.c'};
def.IncPaths = {'rtwdemo_lct_src'};
def.SrcPaths = {'rtwdemo_lct_src'};

```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument 'def', call the function `legacy_code()` again with the first input set to 'sfcn_cmex_generate'. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file `rtwdemo_sfun_cplx_gain.c`.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function `legacy_code()` again with the first input set to 'compile'.

```
legacy_code('compile', def);
```

```

### Start Compiling rtwdemo_sfun_cplx_gain
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_');
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_cplx_gain.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src');
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_cplx_gain
### Exit

```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function `legacy_code()` again with the first input set to 'sfcn_tlc_generate' to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that

includes the S-function, code generation fails. The TLC block file for the S-function is: `rtwdemo_sfncplx_gain.tlc`.

```
legacy_code('sfnc_tlc_generate', def);
```

Generate an `rtwmakecfg.m` File for Code Generation

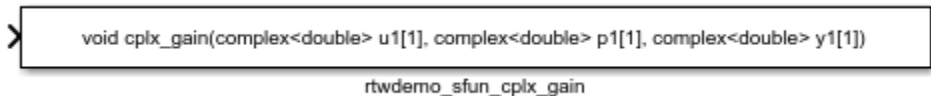
After you create the TLC block file, you can call the function `legacy_code()` again with the first input set to `'rtwmakecfg_generate'` to generate an `rtwmakecfg.m` file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the `rtwmakecfg.m` file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function `legacy_code()` again with the first input set to `'slblock_generate'` to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

```
legacy_code('slblock_generate', def);
```



Show the Integration of the Model with Legacy Code

The model `rtwdemo_lct_cplxgain` shows integration of the model with the legacy code. The subsystem `complex_gain` serves as a harness for the call to the legacy C function via the generate S-function.

```
if isempty(find_system('SearchDepth',0,'Name','rtwdemo_lct_cplxgain'))
    open_system('rtwdemo_lct_cplxgain')
    open_system('rtwdemo_lct_cplxgain/complex_gain')
    sim('rtwdemo_lct_cplxgain')
end
```




Integrate External C Functions That Pass Arguments That Have Inherited Dimensions

This example shows how to use the Legacy Code Tool to integrate legacy C functions whose arguments have inherited dimensions.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional `rtwmakecfg.m` file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototypes of the legacy functions being called in this example are:

- `void mat_add(real_T *u1, real_T *u2, int32_T nbRows, int32_T nbCols, real_T *y1)`
- `void mat_mult(real_T *u1, real_T *u2, int32_T nbRows1, int32_T nbCols1, int32_T nbCols2, real_T *y1)`

`real_T` is a typedef to `double`, and `int32_T` is a typedef to a 32-bit integer. The legacy source code is in the files `mat_ops.h`, and `mat_ops.c`.

```
defs = [];
```

```
% rtwdemo_sfun_mat_add
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_mat_add';
```

```

def.OutputFcnSpec = ['void mat_add(double u1[][], double u2[][], ' ...
                    'int32 u3, int32 u4, double y1[size(u1,1)][size(u1,2)])'];
def.HeaderFiles   = {'mat_ops.h'};
def.SourceFiles   = {'mat_ops.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];

% rtwdemo_sfun_mat_mult
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_mat_mult';
def.OutputFcnSpec = ['void mat_mult(double u1[p1][p2], double u2[p2][p3], ' ...
                    'int32 p1, int32 p2, int32 p3, double y1[p1][p3)'];
def.HeaderFiles   = {'mat_ops.h'};
def.SourceFiles   = {'mat_ops.c'};
def.IncPaths      = {'rtwdemo_lct_src'};
def.SrcPaths      = {'rtwdemo_lct_src'};
defs = [defs; def];

```

Generate S-Functions for Simulation

To generate C-MEX S-functions according to the description provided by the input argument 'defs', call the function `legacy_code()` again with the first input set to 'sfcn_cmex_generate'. The S-functions call the legacy functions during simulation. The source code for the S-function is in the files `rtwdemo_sfun_mat_add.c` and `rtwdemo_sfun_mat_mult.c`.

```
legacy_code('sfcn_cmex_generate', defs);
```

Compile the Generated S-Functions for Simulation

After you generate the C-MEX S-function source files, to compile the S-functions for simulation with Simulink®, call the function `legacy_code()` again with the first input set to 'compile'.

```
legacy_code('compile', defs);
```

```

### Start Compiling rtwdemo_sfun_mat_add
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_mat_add.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src',
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.

```

```

### Finish Compiling rtwdemo_sfun_mat_add
### Exit

### Start Compiling rtwdemo_sfun_mat_mult
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_')
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_mat_mult.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src',
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_mat_mult
### Exit

```

Generate TLC Block Files for Code Generation

After you compile the S-functions and use them in simulation, you can call the function `legacy_code()` again with the first input set to `'sfcn_tlc_generate'` to generate TLC block files. Block files specify how the generated code for a model calls the legacy code. If you do not generate TLC block files and you try to generate code for a model that includes the S-functions, code generation fails. The TLC block files for the S-functions are `rtwdemo_sfun_mat_add.tlc` and `rtwdemo_sfun_mat_mult.tlc`.

```
legacy_code('sfcn_tlc_generate', defs);
```

Generate an `rtwmakecfg.m` File for Code Generation

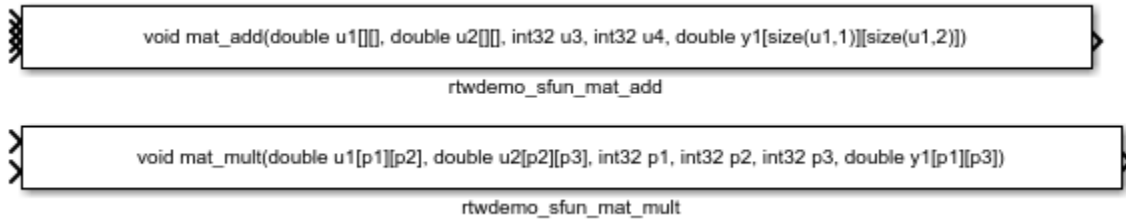
After you create the TLC block files, you can call the function `legacy_code()` again with the first input set to `'rtwmakecfg_generate'` to generate an `rtwmakecfg.m` file to support code generation. If the required source and header files for the S-functions are not in the same folder as the S-functions, and you want to add these dependencies in the makefile produced during code generation, generate the `rtwmakecfg.m` file.

```
legacy_code('rtwmakecfg_generate', defs);
```

Generate Masked S-Function Blocks for Calling the Generated S-Functions

After you compile the C-MEX S-function source, you can call the function `legacy_code()` again with the first input set to `'slblock_generate'` to generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. From there you can copy them to an existing model

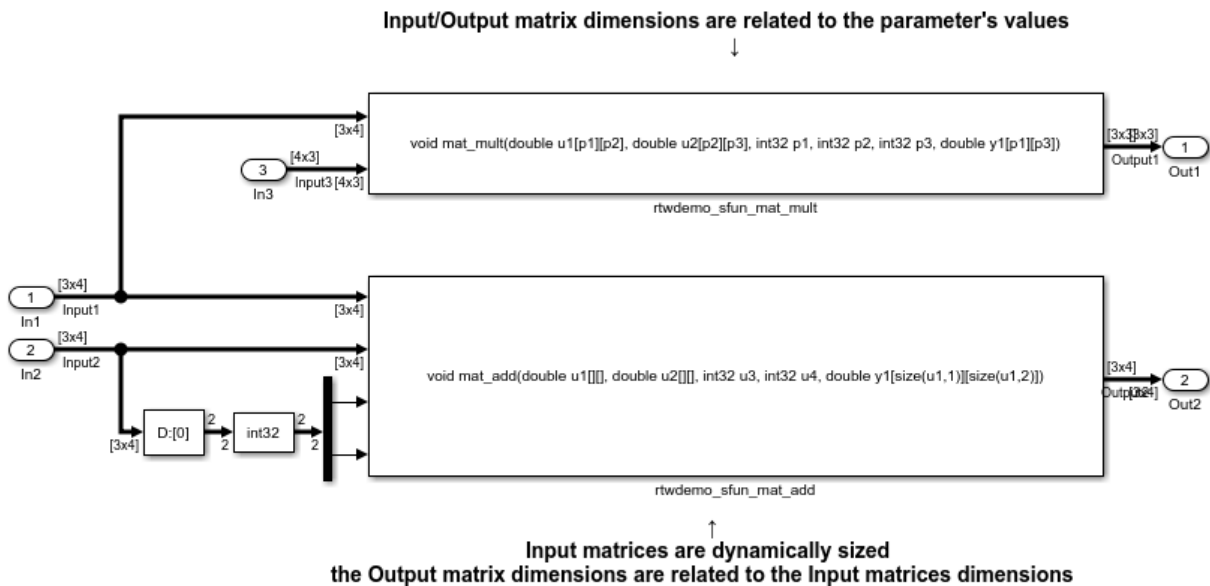
```
legacy_code('slblock_generate', defs);
```



Show the Generated Integration with Legacy Code

The model `rtwdemo_lct_inherit_dims` shows integration of the model with the legacy code. The subsystem `TestMatOps` serves as a harness for the calls to the legacy C functions, with unit delays serving to store the previous output values.

```
open_system('rtwdemo_lct_inherit_dims')  
open_system('rtwdemo_lct_inherit_dims/TestMatOps')  
sim('rtwdemo_lct_inherit_dims')
```



Integrate External C Functions That Implement Start and Terminate Actions

Integrate legacy C functions that have start and terminate actions by using the Legacy Code Tool.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional `rtwmakecfg.m` file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, assign its properties

to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototypes of the legacy functions being called in this example are:

- `void initFaultCounter(unsigned int *counter)`
- `void openLogFile(void **fid)`
- `void incAndLogFaultCounter(void *fid, unsigned int *counter, double time)`
- `void closeLogFile(void **fid)`

The legacy source code is in the files `your_types.h`, `fault.h`, and `fault.c`.

```
% rtwdemo_sfun_fault
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_fault';
def.InitializeConditionsFcnSpec = 'initFaultCounter(uint32 work2[1])';
def.StartFcnSpec = 'openLogFile(void **work1)';
def.OutputFcnSpec = ...
    'incAndLogFaultCounter(void *work1, uint32 work2[1], double u1)';
def.TerminateFcnSpec = 'closeLogFile(void **work1)';
def.HeaderFiles = {'fault.h'};
def.SourceFiles = {'fault.c'};
def.IncPaths = {'rtwdemo_lct_src'};
def.SrcPaths = {'rtwdemo_lct_src'};
def.Options.useTlcWithAccel = false;
```

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument `def`, call the function `legacy_code()` again with the first input set to `'sfcn_cmex_generate'`. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file `rtwdemo_sfun_fault.c`.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function `legacy_code()` again with the first input set to `'compile'`.

```
legacy_code('compile', def);

### Start Compiling rtwdemo_sfun_fault
```

```

    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_fault.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_fault
### Exit

```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function `legacy_code()` again with the first input set to `'sfcn_tlc_generate'` to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: `rtwdemo_sfun_fault.tlc`.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an `rtwmakecfg.m` File for Code Generation

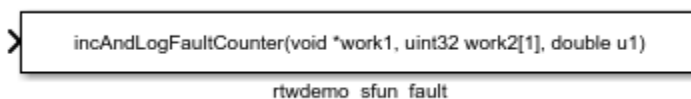
After you create the TLC block file, you can call the function `legacy_code()` again with the first input set to `'rtwmakecfg_generate'` to generate an `rtwmakecfg.m` file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the `rtwmakecfg.m` file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function `legacy_code()` again with the first input set to `'slblock_generate'` to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

```
legacy_code('slblock_generate', def);
```



```

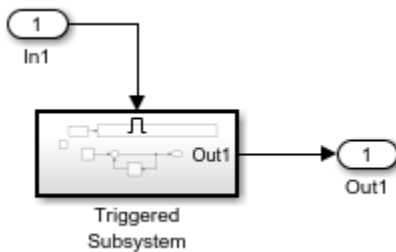
> incAndLogFaultCounter(void *work1, uint32 work2[1], double u1)
    rtwdemo_sfun_fault

```

Showing the Generated Integration with Legacy Code

The model `rtwdemo_lct_start_term` shows integration of the model with the legacy code. The subsystem `TestFixpt` serves as a harness for the call to the legacy C function, and the scope compares the output of the function with the output of the built-in Simulink® product block; the results should be identical.

```
open_system('rtwdemo_lct_start_term')
open_system('rtwdemo_lct_start_term/TestFault')
sim('rtwdemo_lct_start_term')
```



Integrate External C Functions That Pass Arguments as Multi-Dimensional Signals

This example shows how to use the Legacy Code Tool to integrate legacy C functions with multi-dimensional Signals.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional `rtwmakecfg.m` file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, assign its properties

to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototype of the legacy functions being called in this example is:

```
void array3d_add(real_T *y1, real_T *u1, real_T *u2, int32_T nbRows, int32_T nbCols,
int32_T nbPages);
```

`real_T` is a typedef to `double`, and `int32_T` is a typedef to a 32-bit integer. The legacy source code is in the files `ndarray_ops.h`, and `ndarray_ops.c`.

```
% rtwdemo_sfun_ndarray_add
def = legacy_code('initialize');
def.SFunctionName = 'rtwdemo_sfun_ndarray_add';
def.OutputFcnSpec = ['void array3d_add(double y1[size(u1,1)][size(u1,2)][size(u1,3)],
    'double u1[][][], double u2[][][], ' ...
    'int32 size(u1,1), int32 size(u1,2), int32 size(u1,3))'];
def.HeaderFiles = {'ndarray_ops.h'};
def.SourceFiles = {'ndarray_ops.c'};
def.IncPaths = {'rtwdemo_lct_src'};
def.SrcPaths = {'rtwdemo_lct_src'};
```

`y1` is a 3-D output signal of same dimensions as the 3-D input signal `u1`. Note that the last 3 arguments passed to the legacy function correspond to the number of element in each dimension of the 3-D input signal `u1`.

Generate an S-Function for Simulation

To generate a C-MEX S-function according to the description provided by the input argument `def`, call the function `legacy_code()` again with the first input set to `'sfcn_cmex_generate'`. The S-function calls the legacy functions during simulation. The source code for the S-function is in the file `rtwdemo_sfun_ndarray_add.c`.

```
legacy_code('sfcn_cmex_generate', def);
```

Compile the Generated S-Function for Simulation

After you generate the C-MEX S-function source file, to compile the S-function for simulation with Simulink®, call the function `legacy_code()` again with the first input set to `'compile'`.

```
legacy_code('compile', def);
```

```
### Start Compiling rtwdemo_sfun_ndarray_add
```

```

    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_ndarray_add.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_ndarray_add
### Exit

```

Generate a TLC Block File for Code Generation

After you compile the S-function and use it in simulation, you can call the function `legacy_code()` again with the first input set to `'sfcn_tlc_generate'` to generate a TLC block file. The block file specifies how the generated code for a model calls the legacy code. If you do not generate a TLC block file and you try to generate code for a model that includes the S-function, code generation fails. The TLC block file for the S-function is: `rtwdemo_sfun_ndarray_add.tlc`.

```
legacy_code('sfcn_tlc_generate', def);
```

Generate an `rtwmakecfg.m` File for Code Generation

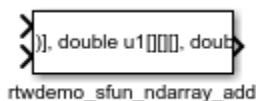
After you create the TLC block file, you can call the function `legacy_code()` again with the first input set to `'rtwmakecfg_generate'` to generate an `rtwmakecfg.m` file to support code generation. If the required source and header files for the S-function are not in the same folder as the S-function, and you want to add these dependencies in the makefile produced during code generation, generate the `rtwmakecfg.m` file.

```
legacy_code('rtwmakecfg_generate', def);
```

Generate a Masked S-Function Block for Calling the Generated S-Function

After you compile the C-MEX S-function source, you can call the function `legacy_code()` again with the first input set to `'slblock_generate'` to generate a masked S-function block that calls that S-function. The software places the block in a new model. From there you can copy it to an existing model.

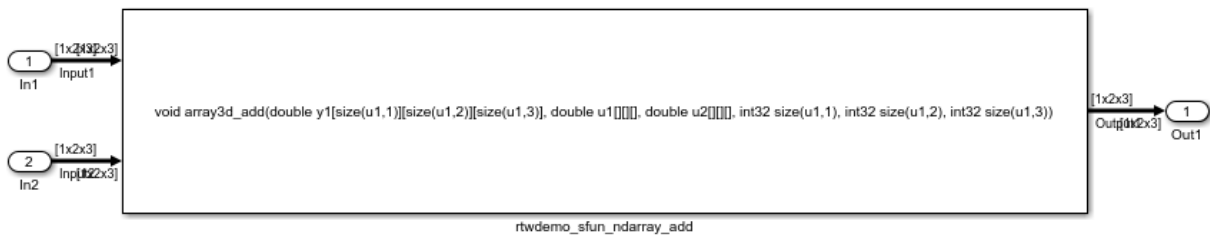
```
legacy_code('slblock_generate', def);
```



Showing the Generated Integration with Legacy Code

The model `rtwdemo_lct_ndarray` shows integration of the model with the legacy code. The subsystem `ndarray_add` serves as a harness for the call to the legacy C function.

```
open_system('rtwdemo_lct_ndarray')
open_system('rtwdemo_lct_ndarray/ndarray_add')
sim('rtwdemo_lct_ndarray')
```



This legacy function computes the addition of the 2 input signals:

- Input1 and Input2 are dynamically sized 3D arrays
- Output1 is a dynamically sized 3D array of same size as Input1
- the last 3 function's arguments allow to pass the Input1's dimensions to the legacy function

Integrate External C Functions with a Block Sample Time Specified, Inherited, and Parameterized

This example shows how to use the Legacy Code Tool to integrate legacy C functions with the block's sample time specified, inherited and parameterized.

With the Legacy Code Tool, you can:

- Provide the legacy function specification.
- Generate a C-MEX S-function that calls the legacy code during simulation.
- Compile and build the generated S-function for simulation.
- Generate a TLC block file and optional `rtwmakecfg.m` file that specifies how the generated code for a model calls the legacy code.

Provide the Legacy Function Specification

Legacy Code Tool functions take a specific data structure or array of structures as the argument. You can initialize the data structure by calling the function `legacy_code()`

using 'initialize' as the first input. After initializing the structure, assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototype of the legacy functions being called in this example is:

```
FLT gainScalar(const FLT in, const FLT gain)
```

FLT is a typedef to float. The legacy source code is in the files `your_types.h`, `gain.h`, and `gainScalar.c`.

```
defs = [];  
  
% rtwdemo_sfun_st_inherited  
def = legacy_code('initialize');  
def.SFunctionName = 'rtwdemo_sfun_st_inherited';  
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';  
def.HeaderFiles = {'gain.h'};  
def.SourceFiles = {'gainScalar.c'};  
def.IncPaths = {'rtwdemo_lct_src'};  
def.SrcPaths = {'rtwdemo_lct_src'};  
defs = [defs; def];  
  
% rtwdemo_sfun_st_fixed  
def = legacy_code('initialize');  
def.SFunctionName = 'rtwdemo_sfun_st_fixed';  
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';  
def.HeaderFiles = {'gain.h'};  
def.SourceFiles = {'gainScalar.c'};  
def.IncPaths = {'rtwdemo_lct_src'};  
def.SrcPaths = {'rtwdemo_lct_src'};  
def.SampleTime = [2 1];  
defs = [defs; def];  
  
% rtwdemo_sfun_st_parameterized  
def = legacy_code('initialize');  
def.SFunctionName = 'rtwdemo_sfun_st_parameterized';  
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';  
def.HeaderFiles = {'gain.h'};  
def.SourceFiles = {'gainScalar.c'};  
def.IncPaths = {'rtwdemo_lct_src'};  
def.SrcPaths = {'rtwdemo_lct_src'};  
def.SampleTime = 'parameterized';  
defs = [defs; def];
```

Generate S-Functions for Simulation

To generate C-MEX S-functions according to the description provided by the input argument 'defs', call the function `legacy_code()` again with the first input set to 'sfcn_cmex_generate'. The S-functions call the legacy functions during simulation. The source code for the S-functions is in the files `rtwdemo_sfun_st_inherited.c` and `rtwdemo_sfun_st_fixed.c`, `rtwdemo_sfun_st_parameterized.c`.

```
legacy_code('sfcn_cmex_generate', defs);
```

Compile the Generated S-Functions for Simulation

After you generate the C-MEX S-function source files, to compile the S-functions for simulation with Simulink®, call the function `legacy_code()` again with the first input set to 'compile'.

```
legacy_code('compile', defs);
```

```
### Start Compiling rtwdemo_sfun_st_inherited
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_1')
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_st_inherited.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_1')
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_st_inherited
### Exit

### Start Compiling rtwdemo_sfun_st_fixed
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_1')
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_st_fixed.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_1')
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
### Finish Compiling rtwdemo_sfun_st_fixed
### Exit

### Start Compiling rtwdemo_sfun_st_parameterized
    mex('-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_1')
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
    mex('rtwdemo_sfun_st_parameterized.c', '-IB:\matlab\toolbox\rtw\rtwdemos\rtwdemo_lct_src', '-IC:\TEMP\Bdoc16b_437511_1')
```

```
Building with 'Microsoft Visual C++ 2013 Professional (C)'.  
MEX completed successfully.  
### Finish Compiling rtwdemo_sfun_st_parameterized  
### Exit
```

Generate TLC Block Files for Code Generation

After you compile the S-functions and use them in simulation, you can call the function `legacy_code()` again with the first input set to `'sfcn_tlc_generate'` to generate TLC block files. Block files specify how the generated code for a model calls the legacy code. If you do not generate TLC block files and you try to generate code for a model that includes the S-functions, code generation fails. The TLC block files for the S-functions are `rtwdemo_sfun_st_inherited.tlc` and `rtwdemo_sfun_st_fixed.tlc`. `rtwdemo_sfun_st_parameterized.tlc`.

```
legacy_code('sfcn_tlc_generate', defs);
```

Generate an `rtwmakecfg.m` File for Code Generation

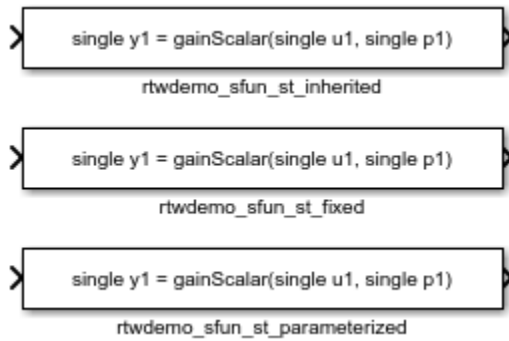
After you create the TLC block files, you can call the function `legacy_code()` again with the first input set to `'rtwmakecfg_generate'` to generate an `rtwmakecfg.m` file to support code generation. If the required source and header files for the S-functions are not in the same folder as the S-functions, and you want to add these dependencies in the makefile produced during code generation, generate the `rtwmakecfg.m` file.

```
legacy_code('rtwmakecfg_generate', defs);
```

Generate Masked S-Function Blocks for Calling the Generated S-Functions

After you compile the C-MEX S-function source, you can call the function `legacy_code()` again with the first input set to `'slblock_generate'` to generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. From there you can copy them to an existing model.

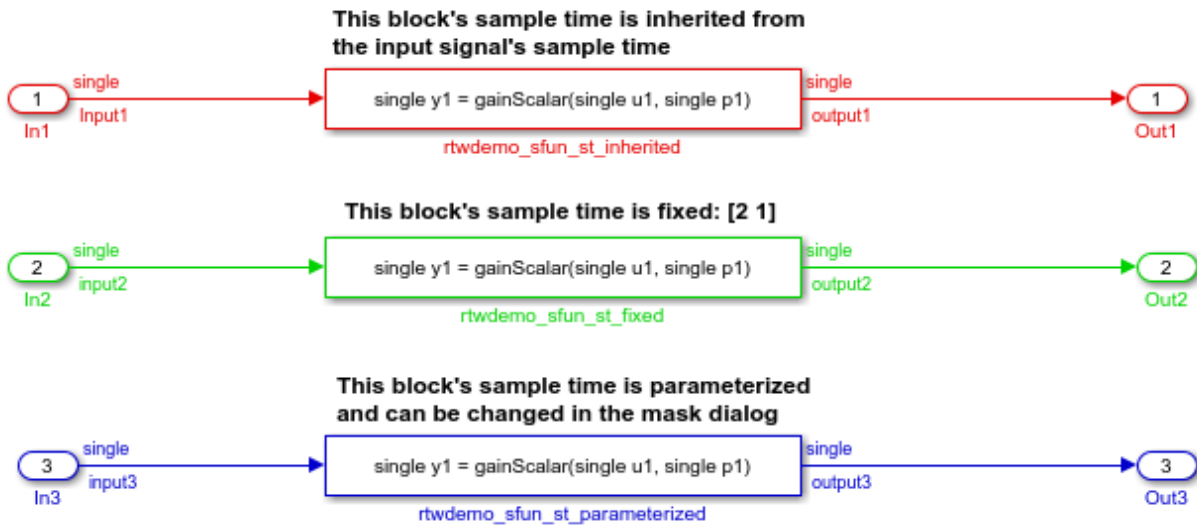
```
legacy_code('slblock_generate', defs);
```



Show the Generated Integration with Legacy Code

The model `rtwdemo_lct_sampletime` shows integration of the model with the legacy code. The subsystem `sample_time` serves as a harness for the calls to the legacy C functions, with unit delays serving to store the previous output values.

```
open_system('rtwdemo_lct_sampletime')
open_system('rtwdemo_lct_sampletime/sample_time')
sim('rtwdemo_lct_sampletime')
```

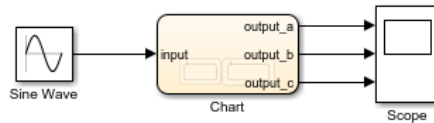


Insert External C and C++ Code Into Stateflow Charts for Code Generation

This example shows how to use Stateflow® to integrate external code into a model.

Open Model

```
model='rtwdemo_sfcustom';
open_system(model);
```

Integrating c-code into model

Custom written c-files can be easily integrated into Stateflow models. This code can be used to augment Stateflow's capabilities or to take advantage of legacy code.

To add custom c-files open the Stateflow simulation target and enter the following:

Include Code - Header that defines functions, structures, and data to be accessible by Stateflow.

Include Path - Path to this include file.

Source Files - c-files that define the functions and data accessible by Stateflow.

▶ [Open Stateflow simulation custom code settings](#)

If generating code via Simulink Coder, these same settings must also be added to the Model's configuration parameters custom code settings.

▶ [Open Code Generation custom code settings](#)

▶ [Build model using Simulink Coder](#)

Calling c-code from Stateflow

Functions

Custom c-code functions can be called from Stateflow using the same syntax as graphical function calls. The statement takes the form:

```
result = my_custom_function(in_args);
```

Structures

Variables of structure type can be accessed in Stateflow via the "dot" notation. The expression takes the form:

```
result = my_var.my_field;
```

To view the custom source for this exemplenstration double click the links below.

▶ [Open my_header.h](#)

▶ [Open my_function.c](#)

Calling C++ code from Stateflow

Custom C++ code can also be integrated into Stateflow and Simulink Coder. Double-click below for an example.

▶ [Open rtwdemo_sfcpp](#)

Additional documentation

Additional documentation is available for integrating C and C++ code in your model by double-clicking the link below.

▶ [C-Code integration](#)

▶ [C++-Code integration](#)

Copyright 1994-2012 The MathWorks, Inc.

Integrate Code

1. The example includes the custom header file `my_header.c` and the custom source file `my_function.c`.

```
%Open files my_header.h and my_function.c
eval('edit my_header.h')
eval('edit my_function.c')
```

2. On the Configuration Parameters dialog box **Simulation Target** pane, enter the custom source file and header file. Also enter additional include directories and source files.

In this example, the custom header file `my_header.c` and source file `my_function.c` are entered on the **Simulation Target** pane.

```
%Open Configuration Parameters dialog box
slCfgPrmDlg(model, 'Open');
slCfgPrmDlg(bdroot, 'TurnToPage', 'Simulation Target');
```

3. If you generate code with Simulink Coder®, on the Configuration Parameters dialog box **Code Generation > Custom Code** pane, enter the same custom source file and header file. Also enter the same additional include directories and source files.

In this example, the custom header file `my_header.c` and source file `my_function.c` are entered on the **Code Generation > Custom Code** pane.

```
%Open Configuration Parameters dialog box
slCfgPrmDlg(model, 'Open');
slCfgPrmDlg(bdroot, 'TurnToPage', 'Code Generation/Custom Code');
```

Generate Code

```
rtwbuild('rtwdemo_sfcustom')

### Starting build procedure for model: rtwdemo_sfcustom
### Successful completion of build procedure for model: rtwdemo_sfcustom
```

Call C Code from Stateflow

To call custom C code functions from Stateflow, use the same syntax as graphical function calls: `result = my_custom_function(in_args);`

To call variables of structure type, use the dot notation: `result = my_var.my_field;`

See Also

- Include Custom C Code in Simulation Targets for Library Models
- Integrate Custom C++ Code for Simulation

Close Model

```
rtwdemoclean;
close_system('rtwdemo_sfcustom',0);
```

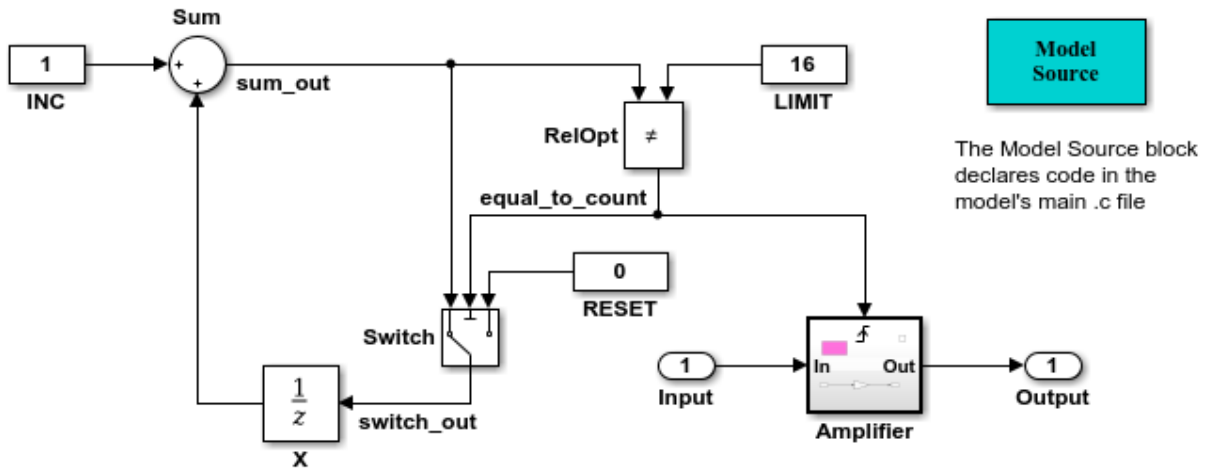
Integrate External C Code Into Generated Code Using Custom Code Blocks and Model Configuration Parameters

The model shows how to integrate external code into generated code by using Simulink® Coder™ custom code blocks and model configuration parameters.

Open Example Model

Open the example model `rtwdemo_slcustcode`.

```
open_system('rtwdemo_slcustcode')
```



Several techniques exist for incorporating custom code into Simulink Coder. This model shows the use of the Simulink Coder custom code blocks and the Configuration Parameters Code Generation Custom Code page:

1. The Model Source custom code block declares an integer GLOBAL_INT1 in <model>.c.
2. The Subsystem Outputs custom code block (inside subsystem Amplifier) uses GLOBAL_INT1.
3. The variable GLOBAL_INT2 is declared and set from the Configuration Parameters Code Generation Custom Code page, from the "Source file" and "Initialize function," respectively.

Some overlap exists between custom code blocks and custom code specified using configuration parameters, but custom code blocks provide much finer granularity of code placement, and have the advantage of being graphical.

Generate Code Using Simulink Coder (double-click)	Generate Code Using Embedded Coder (double-click)	View Custom Code Configuration (double-click)	View Custom Code Library (double-click)
---	---	---	---

Several techniques exist for integrating external code into code that Simulink® Coder™ generates for a model. This model shows how to integrate external C code by using Simulink® Coder™ custom code blocks and model configuration parameters.

In the model:

- Custom code block Model Source declares integer GLOBAL_INT1. The declaration appears in the generated file `model.c`.
- Triggered subsystem Amplifier includes the custom code block System Outputs, which uses GLOBAL_INT1. The System Outputs block declares code for the nearest parent atomic subsystem. In this case, the code appears in the output for the triggered subsystem.
- Model configuration parameters **Source file** and **Initialize function**, on the **Code Generation > Custom Code** pane, declare and set the variable GLOBAL_INT2.

Some overlap exists between custom code blocks and the custom code model configuration parameters. Custom code blocks provide finer granularity of code placement and are graphical.

To view the custom code blocks, double click **View Custom Code Library**. To view the custom code configuration parameters, double click **View Custom Code Configuration**.

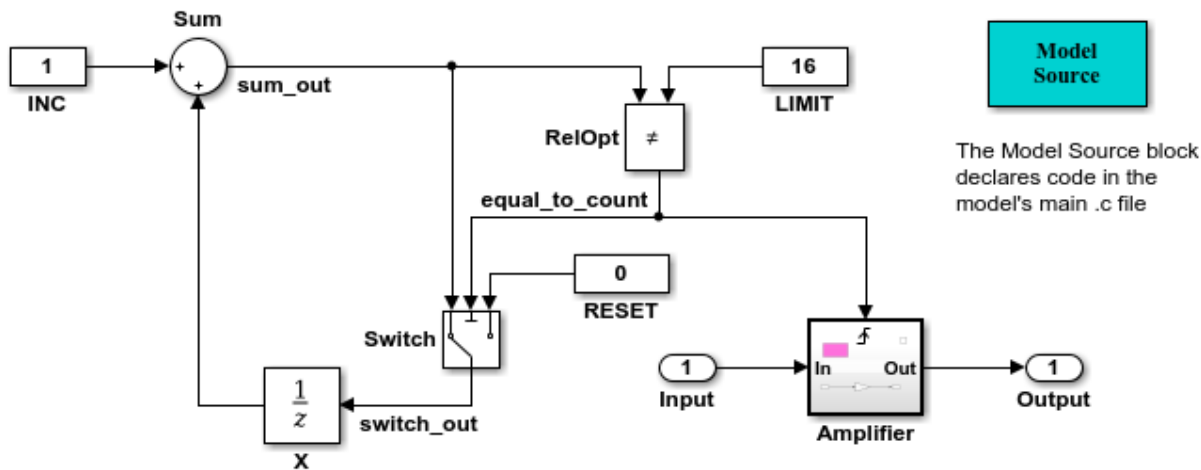
Integrate External C Code Into Generated Code Using Custom Code Blocks and Model Configuration Parameters

The model shows how to integrate external code into generated code by using Simulink® Coder™ custom code blocks and model configuration parameters.

Open Example Model

Open the example model `rtwdemo_slcustcode`.

```
open_system('rtwdemo_slcustcode')
```



Several techniques exist for incorporating custom code into Simulink Coder. This model shows the use of the Simulink Coder custom code blocks and the Configuration Parameters Code Generation Custom Code page:

1. The Model Source custom code block declares an integer GLOBAL_INT1 in <model>.c.
2. The Subsystem Outputs custom code block (inside subsystem Amplifier) uses GLOBAL_INT1.
3. The variable GLOBAL_INT2 is declared and set from the Configuration Parameters Code Generation Custom Code page, from the "Source file" and "Initialize function," respectively.

Some overlap exists between custom code blocks and custom code specified using configuration parameters, but custom code blocks provide much finer granularity of code placement, and have the advantage of being graphical.

<p>Generate Code Using Simulink Coder (double-click)</p>	<p>Generate Code Using Embedded Coder (double-click)</p>	<p>View Custom Code Configuration (double-click)</p>	<p>View Custom Code Library (double-click)</p>
---	---	---	---

Several techniques exist for integrating external code into code that Simulink® Coder™ generates for a model. This model shows how to integrate external C code by using Simulink® Coder™ custom code blocks and model configuration parameters.

In the model:

- Custom code block Model Source declares integer GLOBAL_INT1. The declaration appears in the generated file `model.c`.
- Triggered subsystem Amplifier includes the custom code block System Outputs, which uses GLOBAL_INT1. The System Outputs block declares code for the nearest parent atomic subsystem. In this case, the code appears in the output for the triggered subsystem.
- Model configuration parameters **Source file** and **Initialize function**, on the **Code Generation > Custom Code** pane, declare and set the variable GLOBAL_INT2.

Some overlap exists between custom code blocks and the custom code model configuration parameters. Custom code blocks provide finer granularity of code placement and are graphical.

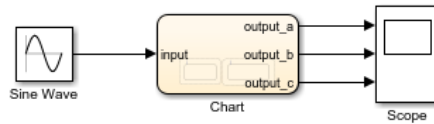
To view the custom code blocks, double click **View Custom Code Library**. To view the custom code configuration parameters, double click **View Custom Code Configuration**.

Insert External C and C++ Code Into Stateflow Charts for Code Generation

This example shows how to use Stateflow® to integrate external code into a model.

Open Model

```
model='rtwdemo_sfcustom';  
open_system(model);
```



Integrating c-code into model

Custom written c-files can be easily integrated into Stateflow models. This code can be used to augment Stateflow's capabilities or to take advantage of legacy code.

To add custom c-files open the Stateflow simulation target and enter the following:

Include Code - Header that defines functions, structures, and data to be accessible by Stateflow.

Include Path - Path to this include file.

Source Files - c-files that define the functions and data accessible by Stateflow.

▶ **Open Stateflow simulation custom code settings**

If generating code via Simulink Coder, these same settings must also be added to the Model's configuration parameters custom code settings.

▶ **Open Code Generation custom code settings**

▶ **Build model using Simulink Coder**

Calling c-code from Stateflow

Functions

Custom c-code functions can be called from Stateflow using the same syntax as graphical function calls. The statement takes the form:

```
result = my_custom_function(in_args);
```

Structures

Variables of structure type can be accessed in Stateflow via the "dot" notation. The expression takes the form:

```
result = my_var.my_field;
```

To view the custom source for this exemplenstration double click the links below.

▶ **Open my_header.h**

▶ **Open my_function.c**

Calling C++ code from Stateflow

Custom C++ code can also be integrated into Stateflow and Simulink Coder. Double-click below for an example.

▶ **Open rtwdemo_sfcpp**

Additional documentation

Additional documentation is available for integrating C and C++ code in your model by double-clicking the link below.

▶ **C-Code integration**

▶ **C++-Code integration**

Copyright 1994-2012 The MathWorks, Inc.

Integrate Code

1. The example includes the custom header file `my_header.c` and the custom source file `my_function.c`.

```
%Open files my_header.h and my_function.c
eval('edit my_header.h')
eval('edit my_function.c')
```

2. On the Configuration Parameters dialog box **Simulation Target** pane, enter the custom source file and header file. Also enter additional include directories and source files.

In this example, the custom header file `my_header.c` and source file `my_function.c` are entered on the **Simulation Target** pane.


```
%Open Configuration Parameters dialog box
slCfgPrmDlg(model, 'Open');
slCfgPrmDlg(bdroot, 'TurnToPage', 'Simulation Target');
```

3. If you generate code with Simulink Coder®, on the Configuration Parameters dialog box **Code Generation > Custom Code** pane, enter the same custom source file and header file. Also enter the same additional include directories and source files.

In this example, the custom header file `my_header.c` and source file `my_function.c` are entered on the **Code Generation > Custom Code** pane.

```
%Open Configuration Parameters dialog box
slCfgPrmDlg(model, 'Open');
slCfgPrmDlg(bdroot, 'TurnToPage', 'Code Generation/Custom Code');
```

Generate Code

```
rtwbuild('rtwdemo_sfcustom')

### Starting build procedure for model: rtwdemo_sfcustom
### Successful completion of build procedure for model: rtwdemo_sfcustom
```

Call C Code from Stateflow

To call custom C code functions from Stateflow, use the same syntax as graphical function calls: `result = my_custom_function(in_args);`

To call variables of structure type, use the dot notation: `result = my_var.my_field;`

See Also

- Include Custom C Code in Simulation Targets for Library Models
- Integrate Custom C++ Code for Simulation

Close Model

```
rtwdemoclean;
close_system('rtwdemo_sfcustom',0);
```

More About

- “Integrate External C Functions That Pass Input and Output Arguments as Parameters with a Fixed-Point Data Type”
- “Integrate External C Functions That Implement N-Dimensional Table Lookups”

- “Integrate External C++ Object Methods”

Configure Model for External Code Integration

Configure a model using the **Custom Code** pane such that the code generator includes external code such as headers, files, and functions. You can also include additional files and paths in the build process.

To...	Select...
Insert custom code near the top of the generated <i>model.c</i> or <i>model.cpp</i> file, outside of any function	<p>Source file and enter the custom code to insert.</p> <hr/> <p>Note: If you generate subsystem code into separate files, this code does not have access to custom code specified as a Source file setting. For example, if you specify an include file as a Source file setting, the software inserts the <code>#include</code> near the top of the <i>model.c</i> or <i>model.cpp</i> file. Subsystem code generated to a separate file does not have access to declarations inside that include file. In this case, consider specifying your custom code as a Header file setting instead.</p>
Insert custom code near the top of the generated <i>model.h</i> file	Header file and enter the custom code to insert.
Insert custom code inside the model's initialize function in the <i>model.c</i> or <i>model.cpp</i> file	Initialize function
Insert custom code inside the model's terminate function in the <i>model.c</i> or <i>model.cpp</i> file.	Terminate function and enter the custom code to insert. Also select the Terminate function required parameter on the Interface pane.
Add include folders, which contain header files, to the build process	Include directories and enter the absolute or relative paths to the folders. If you specify relative paths, the paths must be relative to the folder containing your model files, not relative to the build folder. The order in which you specify the folders is the order in which they are searched for header, source, and library files.
Add source files to be compiled and linked	Source files and enter the full paths or just the file names for the files. Enter just the file name if the file is in the current MATLAB folder or in

To...	Select...
	<p>one of the include folders. For each additional source that you specify, the build process expands a generic rule in the template makefile for the folder in which the source file is found. For example, if a source file is found in folder <code>inc</code>, the build process adds a rule similar to the following:</p> <pre>%.obj: builddir\inc\%.c \$(CC) -c -Fo\$(@F) \$(CFLAGS) \$<</pre> <p>The build process adds the rules in the order you list the source files.</p>
Add libraries to be linked	Libraries and enter the full paths or just the file names for the libraries. Enter just the file name if the library is located in the current MATLAB folder or in one of the include folders.
Add preprocessor macro definitions	Defines and enter a space-separated list of preprocessor macro definitions to be added to the generated code. The list can include both simple definitions (for example, <code>-DEF1</code>) and definitions with a value (for example, <code>-DDEF2=1</code>). Definitions may omit the <code>-D</code> (for example, <code>-DF00=1</code> and <code>F00=1</code> are equivalent). If used, the <code>-D</code> may be overridden by the toolchain if it uses a different flag for defines.
Use the same custom code settings as those specified for simulation of MATLAB Function blocks, Stateflow charts, and Truth Table blocks	<p>Use the same custom code settings as Simulation Target</p> <hr/> <p>Note: This option refers to the Simulation Target pane in the Configuration Parameters dialog box.</p>
Enable a library model to use custom code settings unique from the parent model to which the library is linked	<p>Use local custom code settings (do not inherit from main model)</p> <hr/> <p>Note: This option is available only for library models that contain MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</p>

If you are including a header file, in your custom header file add `#ifndef` code. This avoids multiple inclusions. For example, in `rtwtypes.h` the following `#include` guards are added:

```
#ifndef RTW_HEADER_rtwtypes_h_
#define RTW_HEADER_rtwtypes_h_
```

```
...  
#endif /* RTW_HEADER_rtwtypes_h_ */
```

Note Custom code that you include in a configuration set is ignored when building S-function targets, accelerated simulation targets, and model reference simulation targets.

For more information about **Custom Code** pane parameters, see “Model Configuration Parameters: Code Generation Custom Code”.

More About

- “About External Code Integration” on page 24-2

Insert Code into Model Custom Code Blocks

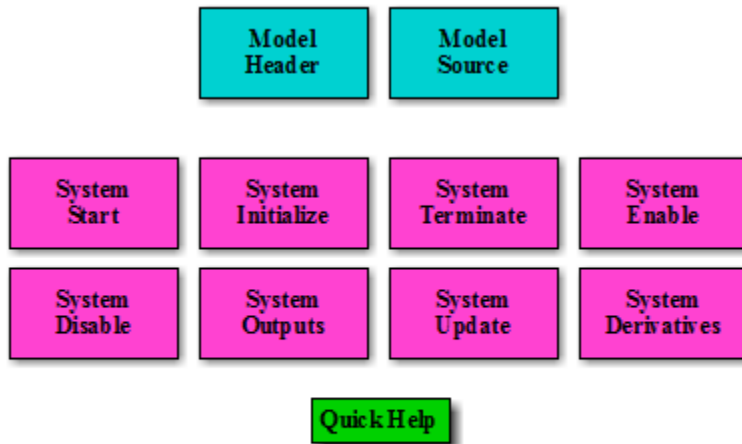
The following topics explain how to use blocks in the Custom Code block library to insert custom code into the code generated for a model.


In this section...
“Custom Code Library” on page 24-86
“Embed Custom Code Directly Into MdlStart Function” on page 24-90
“Custom Code in Subsystems” on page 24-92
“Preserve User Files in Build Folder” on page 24-93

Custom Code Library

The Custom Code library contains blocks that enable you to insert your own C or C++ code into specific functions within code produced by the code generator for root models and subsystems. These blocks are a superset of code customization capabilities built into the **Custom Code** Configuration Parameters dialog box, and provide greater flexibility in terms of code placement than the controls on the dialog box.

The Custom Code library is part of the Simulink Coder block library. You can access the block library by using the Simulink Library Browser. You can access Custom Code blocks by using the Simulink Coder block library or by entering the MATLAB command `rtwlib` and then double-clicking the Custom Code Library block within it. Alternatively, you can enter the command `custcode`.



 These blocks allow you to insert custom code into specific files and functions.

Note: If you need to integrate custom C++ code with generated C code or vice versa, see “Modify External Code for Language Compatibility” on page 24-20.

All Custom Code blocks except for Model Header and Model Source can be dragged into either root models or atomic subsystems. Model Header and Model Source blocks can only be placed in root models.

Note You can use models containing Custom Code blocks as models referenced by Model blocks. The code generator ignores the blocks when producing code for a simulation target. When producing code for a code generation target, the code generator includes and compiles the custom code.

The Custom Code library contains ten blocks that insert custom code into the generated model files and functions. You can view the blocks either by

- Expanding the Custom Code node (under Simulink Coder block library) in the Simulink Library Browser

- Right-clicking the Custom Code sublibrary icon in the right pane of the Simulink Library Browser

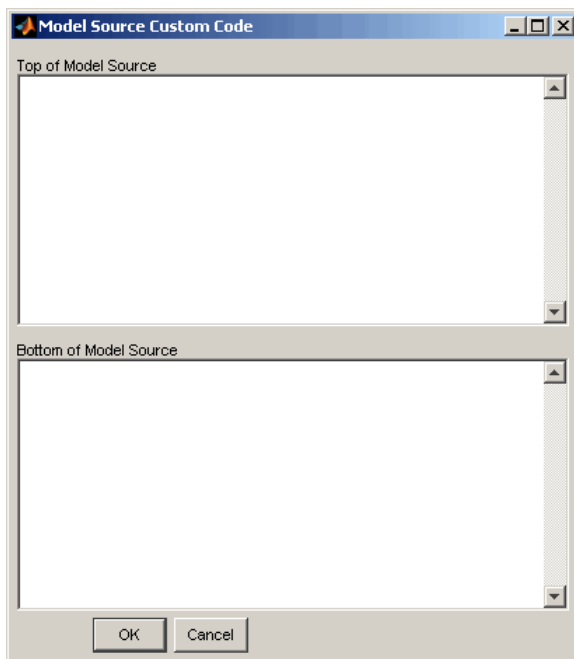
The two blocks on the top row contain text fields for inserting custom code at the top and bottom of

- `model.h` — Model Header File block
- `model.c` or `model.cpp` — Model Source File block

Each block contains two fields, in which you type or paste code and comments:

- Top of Model Source/Header
- Bottom of Model Source/Header

The next figure shows the Model Source block dialog box.

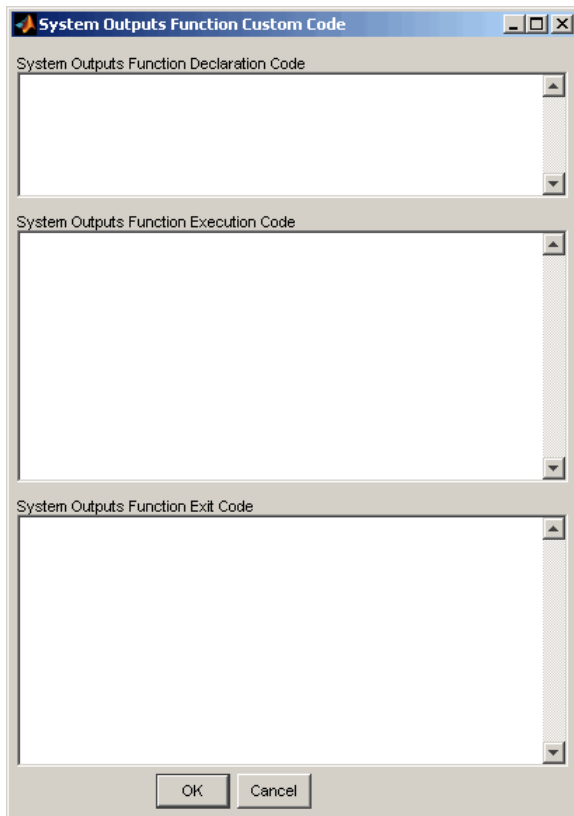


The eight function blocks in the second and third rows contain text fields to insert custom code sections at the top and bottom of these designated model functions:

- SystemStart — System Start function block
- SystemInitialize — System Initialize function block
- SystemTerminate — System Terminate function block
- SystemEnable — System Enable function block
- SystemDisable — System Disable function block
- SystemOutputs — System Outputs function block
- SystemUpdate — System Update function block
- SystemDerivatives — System Derivatives function block

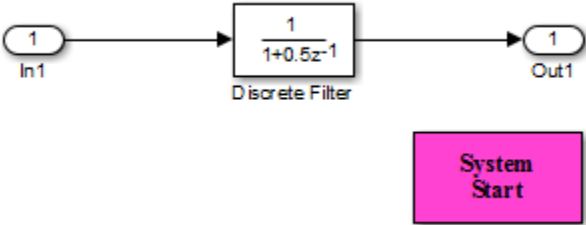
Each of these blocks provides a System Outputs Function Custom Code dialog box that contains three fields:

- Declaration code
- Execution code
- Exit code

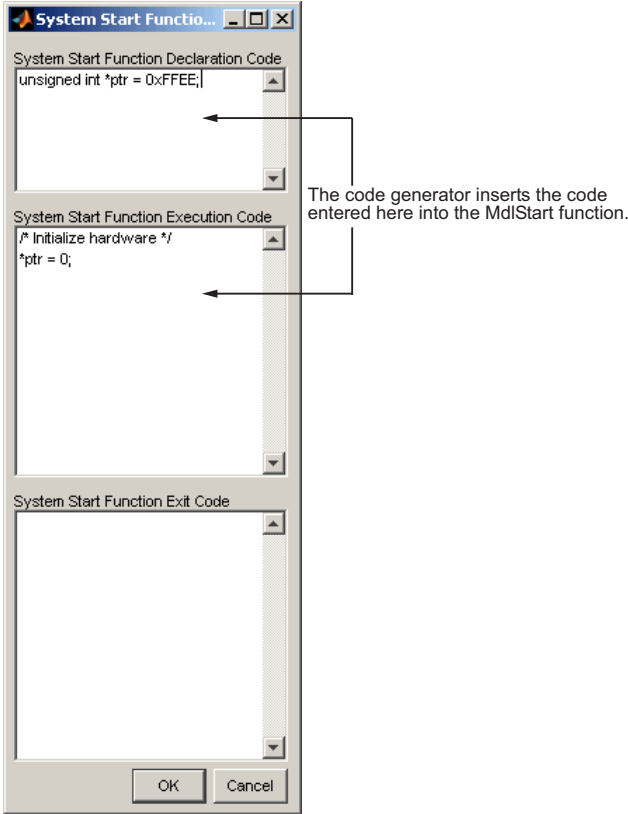


Embed Custom Code Directly Into MdlStart Function

The following example uses a System Start Function block to introduce code into the MdlStart function. The next figure shows a simple model with the System Start Function block inserted.



Double-clicking the System Start Function block opens the System Start Function Custom Code dialog box.



You can insert custom code into the available text fields.

The code below is the `MdlStart` function for this example (`mymodel`).

```
void MdlStart(void)
{
    {
        {
            /* user code (Start function Header) */
            /* System '<Root>' */
            unsigned int *ptr = 0xFFEE;

            /* user code (Start function Body) */
            /* System '<Root>' */
            /* Initialize hardware */
            *ptr = 0;
        }
    }
}

MdlInitialize();
}
```

The custom code entered in the System Start Function Custom Code dialog box is embedded directly in the generated code. Each block of custom code is tagged with a comment such as

```
/* user code (Start function Header) */
```

Custom Code in Subsystems

The location of a Custom Code block in your model determines the location of the code it contains. You can use System Custom Code blocks either at root level or within atomic subsystems; the code is local to the subsystem in which you place the blocks. For example, the System Outputs block places code in `mdlOutputs` when the code block resides in the root model. If the System Outputs block resides in a triggered or enabled subsystem, however, the code is placed in the subsystem's `Outputs` function.

The ordering for a triggered or enabled system is

- 1 Output entry
- 2 Output exit
- 3 Update entry
- 4 Update exit

Note If a root model or atomic subsystem does not need to generate a function for which a Custom Code block has been supplied, either the code in the block is not

used or an error is generated. The software does not have a diagnostic setting to control this. To eliminate the error, remove the Custom Code block.

Preserve User Files in Build Folder

Foreign source files are by default deleted during builds, but can be preserved by following the guidelines given below.

If you put a `.c/.cpp` or `.h` source file in a build folder, and you want to prevent the code generator from deleting it during the TLC code generation process, insert the text `target specific file` in the first line of the `.c/.cpp` or `.h` file. For example,

```
/* COMPANY-NAME target specific file
 *
 * This file is created for use with the
 * COMPANY-NAME target.
 * It is used for ...
 */
...
```

Make sure you spell the text “target specific file” as shown in the preceding example, and that the text is in the first line of the source file. Other text can appear before or after this text.

In addition to preserving them, flagging user files in this manner prevents postprocessing them to indent them along with generated source files. Auto-indenting occurred in previous releases to build folder files with names having the pattern `model_* .c/.cpp` (where `*` could be text). The indenting is harmless, but can cause differences to be detected by source control software that might trigger unnecessary updates.

Insert External Code into Stateflow Charts

“External Code Integration Examples” on page 24-37 includes examples that show how to integrate external code in models for code generation by inserting the code into Stateflow charts.

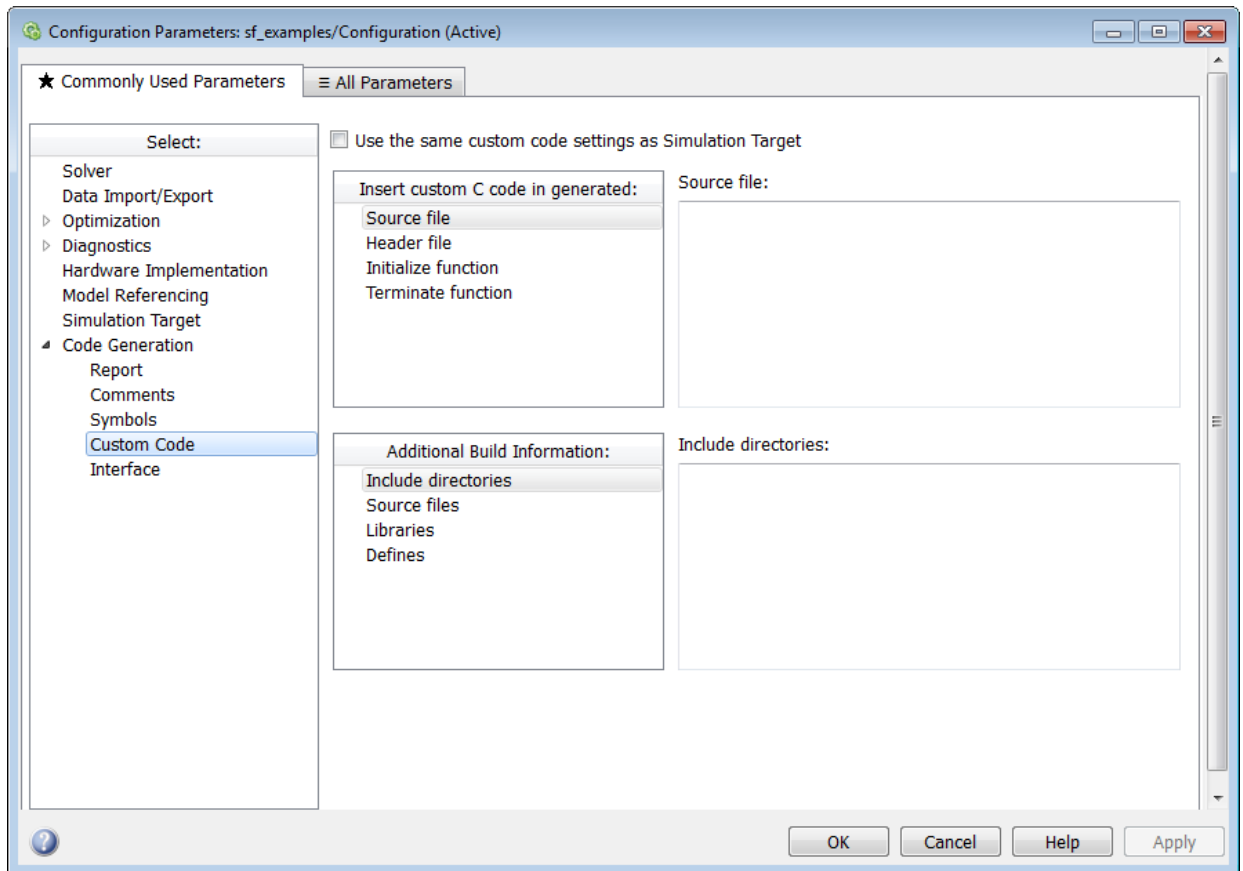
In this section...
“Integrate External Code for Library Charts” on page 24-94
“Integrate External Code for All Charts” on page 24-96

Integrate External Code for Library Charts

To integrate external code that applies only to Stateflow library charts for code generation, complete the following steps for each library model that contributes a chart to your main model. Then, generate code.

- 1 In the Stateflow Editor, select **Code > C/C++ Code > Code Generation Options**.

The Model Configuration Parameters dialog box appears.



- 2 In the **Code Generation** pane, select **Use local custom code settings (do not inherit from main model)**.

This causes each library model to retain its own custom code settings during code generation.

- 3 Specify your custom code in the subpanes.

Follow the guidelines in “Specify Relative Paths for Custom Code”.

Note: If you specified custom code settings for simulation, you can apply these settings to code generation. To avoid entering the same information twice, select **Use the same custom code settings as Simulation Target**.

- 4 Click **OK**.

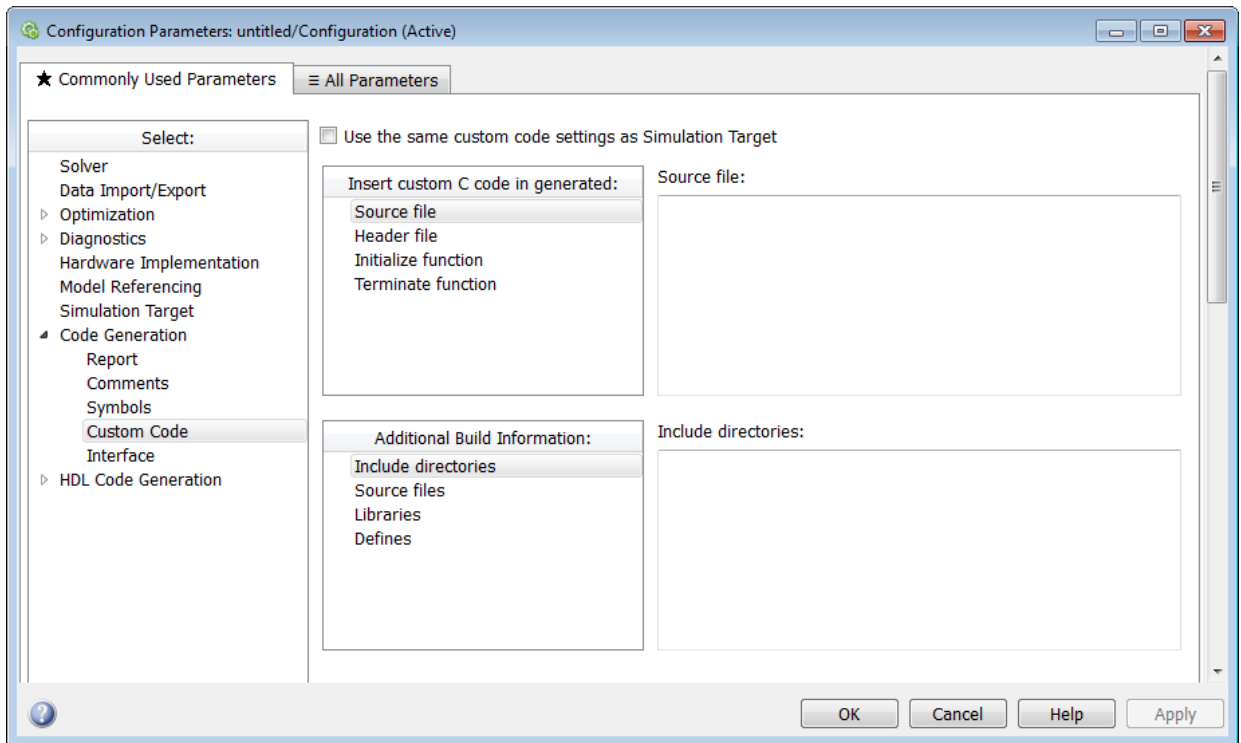
After completing these steps for each library model generate code.

Integrate External Code for All Charts

To integrate external code that applies to all charts for code generation, complete the following steps.

- 1 Specify custom code options for code generation of your main model.
 - a In the Model Configuration Parameters dialog box, select **Code Generation > Custom Code**.

The custom code options appear.



- b** Specify your custom code in the subpanes.

Follow the guidelines in “Specify Relative Paths for Custom Code”.

Note: If you specified custom code settings for simulation, you can apply these settings to code generation. To avoid entering the same information twice, select **Use the same custom code settings as Simulation Target**.

- 2** Configure code generation for each library model that contributes a chart to your main model.
 - a** In the Stateflow Editor, select **Code > C/C++ Code > Code Generation Options**.
 - b** In the **Code Generation** pane, clear the **Use local custom code settings (do not inherit from main model)** check box.

This causes library charts to inherit the custom code settings of your main model.

- c** Click **OK**.
- 3** Generate code.

S-Functions and Code Generation

This topic describes types of S-functions and provides examples of how you can use them with the code generator.

This topic assumes that you understand the following concepts:

- Level 2 S-functions
- Target Language Compiler (TLC) scripting
- How the code generator produces and builds C/C++ code

Note When this topic refers to actions performed by the Target Language Compiler, including parsing, caching, and creating buffers, the name Target Language Compiler is spelled out fully. When referring to code written in the Target Language Compiler syntax, this topic uses the abbreviation TLC.

Note The guidelines presented in this topic are for code generator users. Even if you do not currently use the code generator, follow the practices in this topic when writing S-functions, especially if you are creating general-purpose S-functions.

In this section...

“Classes of Problems Solved by S-Functions” on page 24-99

“Types of S-Functions” on page 24-100

“Basic Files Required for Implementation” on page 24-102

“Guidelines for Writing S-Functions” on page 24-103

Classes of Problems Solved by S-Functions

S-functions help solve various kinds of problems you might face when working with the Simulink and code generator products. These problems include:

- Extending the set of algorithms (blocks) provided by the Simulink and code generator products
- Interfacing external code with the Simulink and code generator products
- Interfacing to hardware through device driver S-functions

- Generating highly optimized code for embedded systems
- Verifying code generated for a subsystem as part of a Simulink simulation

S-functions are written using an application program interface (API) that allows you to implement generic algorithms in the Simulink environment with a great deal of flexibility. This flexibility cannot always be maintained when you use S-functions with the code generator. For example, it is not possible to access the MATLAB workspace from an S-function that is used with the code generator. However, using the techniques presented in this topic, you can create S-functions for most applications that work with the generated code.

Although S-functions provide a generic and flexible solution for implementing complex algorithms in a model, the underlying API incurs overhead in terms of memory and computation resources. Most often the additional resources are acceptable for real-time rapid prototyping systems. In many cases, though, additional resources are unavailable in real-time embedded applications. You can minimize memory and computational requirements by using the Target Language Compiler technology provided with the code generator to inline your S-functions. If you are producing an S-function for existing code, consider using the Legacy Code Tool.

Types of S-Functions

The implementation of S-functions changes based on your requirements. This topic discusses the typical problems that you may face and how to create S-functions for applications that need to work with the Simulink and code generator products. These are some (informally defined) common situations:

- 1 “I’m not concerned with efficiency. I just want to write one version of my algorithm and have it work in the Simulink and code generator products automatically.”
- 2 “I want to implement a highly optimized algorithm in the Simulink and code generator products that looks like a built-in block and generates very efficient code.”
- 3 “I have a lot of hand-written code that I need to interface. I want to call my function from the Simulink and code generator products in an efficient manner.”

MathWorks products have adopted terminology for these different requirements. Respectively, the situations described above map to this terminology:

- 1 Noninlined S-function
- 2 Inlined S-function

3 Autogenerated S-function for legacy or custom code

Noninlined S-Functions

A noninlined S-function is a C or C++ MEX S-function that is treated identically by the Simulink engine and generated code. In general, you implement your algorithm once according to the S-function API. The Simulink engine and generated code call the S-function routines (for example, `mdlOutputs`) during model execution.

Additional memory and computation resources are required for each instance of a noninlined S-Function block. However, this routine of incorporating algorithms into models and code generation applications is typical during the prototyping phase of a project where efficiency is not important. The advantage gained by forgoing efficiency is the ability to change model parameters and structures rapidly.

Writing a noninlined S-function does not involve TLC coding. Noninlined S-functions are the default case for the build process in the sense that once you build a MEX S-function in your model, there is no additional preparation prior to pressing **Ctrl+B** to build your model.

Some restrictions exist concerning the names and locations of noninlined S-function files when generating makefiles. See “Write Noninlined S-Function and TLC Files” on page 24-104.

Inlined S-Functions

For S-functions to work in the Simulink environment, some overhead code is generated. When the code generator produces code from models that contain S-functions (without `sfunction.tlc` files), it embeds some of this overhead code in the generated code. If you want to optimize your real-time code and eliminate some of the overhead code, you must *inline* (or embed) your S-functions. This involves writing a TLC (`sfunction.tlc`) file that eliminates overhead code from the generated code. The Target Language Compiler processes `sfunction.tlc` files to define how to inline your S-function algorithm in the generated code.

Note The term *inline* should not be confused with the C++ *inline* keyword. Inline means to specify text in place of the call to the general S-function API routines (for example, `mdlOutputs`). For example, when a TLC file is used to inline an S-function, the generated code contains the C/ C++ code that would normally appear within the S-function routines and the S-function itself has been removed from the build process.

A fully inlined S-function builds your algorithm (block) into generated code in a manner that is indistinguishable from a built-in block. Typically, a fully inlined S-function requires you to implement your algorithm twice: once for the Simulink model (C/C++ MEX S-function) and once for code generation (TLC file). The complexity of the TLC file depends on the complexity of your algorithm and the level of efficiency you're trying to achieve in the generated code. TLC files vary from simple to complex in structure.

Autogenerated S-Functions for Legacy or Custom Code

If you need to invoke hand-written C/C++ code in your model, consider using the Simulink Legacy Code Tool. The Legacy Code Tool can automate the generation of a fully inlined S-function and a corresponding TLC file based on information that you register in a Legacy Code Tool data structure.

For more information, see “Integrate C Functions Using Legacy Code Tool” in the Simulink Writing S-Functions documentation and “Build S-Function With Legacy Code Tool” on page 24-27.

Basic Files Required for Implementation

This topic briefly describes what files and functions you need to create noninlined and inlined S-functions.

- Noninlined S-functions require the C or C++ MEX S-function source code (*sfunction.c* or *sfunction.cpp*).
- Fully inlined S-functions require an *sfunction.tlc* file. Fully inlined S-functions produce the optimal code for a parameterized S-function. This is an S-function that operates in a specific mode dependent upon fixed S-function parameters that do not change during model execution. For a given operating mode, the *sfunction.tlc* file specifies the exact code that is generated to implement the algorithm for that mode. For example, the direct-index lookup table S-function in “Write Fully Inlined S-Functions with mdlRTW Routine” on page 24-116 contains two operating modes — one for evenly spaced *x-data* and one for unevenly spaced *x-data*.

Note: Fully-inlined S-functions that are generated to invoke legacy or custom C/C++ code also require an *sfunction.tlc* file, which is generated by Legacy Code Tool.

Fully inlined S-functions might require the placement of the mdlRTW routine in your S-function MEX-file *sfunction.c* or *sfunction.cpp*. The mdlRTW routine lets you

place information in *model.rtw*, the record file that specifies a model, and which the code generator invokes the Target Language Compiler to process prior to executing *sfunction.tlc* when generating code.

Including a `mdlRTW` routine is useful when you want to introduce nontunable parameters into your TLC file. Such parameters are generally used to determine which operating mode is active in a given instance of the S-function. Based on this information, the TLC file for the S-function can generate highly efficient, optimal code for that operating mode.

Guidelines for Writing S-Functions

- You can use C/C++ MEX, MATLAB language, and Fortran MEX S-functions with code generation.
- You can inline S-functions for code generation by providing an inlining TLC file. See S-Function Inlining in “Target Language Compiler”. MATLAB and Fortran MEX S-functions must be inlined. C/C++ MEX S-functions can be inlined for code efficiency, or noninlined.
- To automatically generate a fully inlined C MEX S-function for invoking legacy or custom code, use the Legacy Code Tool. For more information, see “Integrate C Functions Using Legacy Code Tool” in the Simulink documentation and “Build S-Function With Legacy Code Tool” on page 24-27.
- If code efficiency is not an overriding consideration, for example, if you are rapid prototyping, you can choose not to inline a C/C++ MEX S-function. For more information, see “Write Noninlined S-Function and TLC Files” on page 24-104.

See Also

`legacy_code`

Related Examples

- “Integrate C Functions Using Legacy Code Tool”
- “Build S-Function With Legacy Code Tool” on page 24-27
- “Inlining S-Functions”

Write S-Function and TLC Files By Hand

You can choose from several approaches for writing S-function and TLC files by hand.

Write Noninlined S-Function and TLC Files

- “About Noninlined S-Functions” on page 24-104
- “Guidelines for Writing Noninlined S-Functions” on page 24-104
- “Noninlined S-Function Parameter Type Limitations” on page 24-105

About Noninlined S-Functions

Noninlined S-functions are identified by the *absence* of an `sfunction.tlc` file for your S-function. The filename varies depending on your platform. For example, on a 64-bit Microsoft Windows system, the file name would be `sfunction.mexw64`. Type `mexext` in the MATLAB Command Window to see which extension your system uses.

Guidelines for Writing Noninlined S-Functions

- The MEX-file cannot call MATLAB functions.
- If the MEX-file uses functions in the MATLAB External Interface libraries, include the header file `cg_sfun.h` instead of `mex.h` or `simulink.c`. To handle this case, include the following lines at the end of your S-function:

```
#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif
```

- Use only MATLAB API function that the code generator supports, which include:

```
mxGetEps
mxGetInf
mxGetM
mxGetN
mxGetNaN
mxGetPr
mxGetScalar
mxGetString
mxIsEmpty
mxIsFinite
mxIsInf
```


- MEX library calls are not supported in generated code. To use such calls in MEX-file and not in the generated code, conditionalize the code as follows:

```
#ifdef MATLAB_MEX_FILE
#endif
```

- Use only full matrices that contain only real data.
- Do not specify a return value for calls to `mxGetString`. If you do specify a return value, the MEX-file will not compile. Instead, use the function's second input argument, which returns a pointer to a character vector.
- Make sure that the `#define s-function_name` statement is correct. The S-function name that you specify must match the S-function's filename.
- Use the data types `real_T` and `int_T` instead of `double` and `int`, if possible. The data types `real_T` and `int_T` are more generic and can be used in multiple environments.
- Provide the build process with the names of the modules used to build the S-function. You can do this by using a template make file, the `set_param` function, or the `S-function modules` field of the S-Function block parameters dialog box. For example, suppose you build your S-function with the following command:

```
mex sfun_main.c sfun_module1.c sfun_module2.c
```

You can then use the following call to `set_param` to include the required modules:

```
set_param(sfun_block, "SFunctionModules", "sfun_module1 sfun_module2")
```

When you are ready to generate code, force the code generator to rebuild the top model, as explained in “Control Regeneration of Top Model Code” on page 25-43.

Noninlined S-Function Parameter Type Limitations

Parameters to noninlined S-functions can be of the following types only:

- Double precision
- Characters in scalars, vectors, or 2-D matrices

For more flexibility in the type of parameters you can supply to S-functions or the operations in the S-function, inline your S-function and consider using an `mdlRTW` S-function routine.

Use of other functions from the MATLAB `matrix.h` API or other MATLAB APIs, such as `mex.h` and `mat.h`, is not supported. If you call unsupported APIs from an S-function

source file, compiler errors occur. See the files `matlabroot/rtw/c/src/rt_matrix.h` and `matlabroot/rtw/c/src/rt_matrix.c` for details on supported MATLAB API functions.

If you use `mxGetPr` on an empty matrix, the function does not return `NULL`; rather, it returns a random value. Therefore, you should protect calls to `mxGetPr` with `mxIsEmpty`.

Write Wrapper S-Function and TLC Files

This topic describes how to create S-functions that work seamlessly with the Simulink and code generator products using the *wrapper* concept. This topic begins by describing how to interface your algorithms in Simulink models by writing MEX S-function wrappers (`sfunction.mex`). It finishes with a description of how to direct the code generator to insert your algorithm into the generated code by creating a TLC S-function wrapper (`sfunction.tlc`).

- “MEX S-Function Wrapper” on page 24-106
- “TLC S-Function Wrapper” on page 24-111
- “Code Overhead for Noninlined S-Functions” on page 24-111
- “How to Inline” on page 24-112
- “The Inlined Code” on page 24-114

MEX S-Function Wrapper

Creating S-functions using an S-function wrapper allows you to insert C/C++ code algorithms in Simulink models and the generated code with little or no change to your original C/C++ function. A *MEX S-function wrapper* is an S-function that calls code that resides in another module. A *TLC S-function wrapper* is a TLC file that specifies how the code generator should call your code (the same code that was called from the C MEX S-function wrapper).

Note: A MEX S-function wrapper must only be used in the MATLAB version in which the wrapper is created.

Suppose you have an algorithm (that is, a C function) called `my_alg` that resides in the file `my_alg.c`. You can integrate `my_alg` into a Simulink model by creating a MEX S-

function wrapper (for example, `wrapsfcn.c`). Once this is done, a Simulink model can call `my_alg` from an S-Function block. However, the Simulink S-function contains a set of empty functions that the Simulink engine requires for various API-related purposes. For example, although only `mdlOutputs` calls `my_alg`, the engine calls `mdlTerminate` as well, even though this S-function routine performs no action.

You can integrate `my_alg` into generated code (that is, embed the call to `my_alg` in the generated code) by creating a TLC S-function wrapper (for example, `wrapsfcn.tlc`). The advantage of creating a TLC S-function wrapper is that the empty function calls can be eliminated and the overhead of executing the `mdlOutputs` function and then the `my_alg` function can be eliminated.

Wrapper S-functions are useful when you are creating new algorithms that are procedural in nature or when you are integrating legacy code into a Simulink model. However, if you want to create code that is

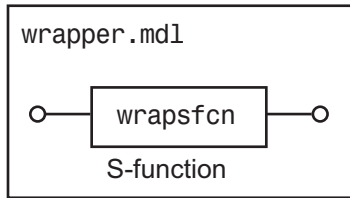
- Interpretive in nature (that is, highly parameterized by operating modes)
- Heavily optimized (that is, no extra tests to decide what mode the code is operating in)

then you must create a *fully inlined TLC file* for your S-function.

The next figure shows the wrapper S-function concept.

Simulink

Place the name of your S-function in the S-Function block dialog box.



In Simulink, the S-function calls mdlOutputs, which in turn calls my_alg.

```

wrapsfcn.c
...
mdlOutputs(...)
{
    ...
    my_alg();
}
    
```

mdlOutputs in wrapsfcn.mex calls external function my_alg.

```

my_alg.c
...
real_T my_alg(real_T u)
{
    ...
    y=f(u);
}
    
```

Simulink Coder

wrapper.c, the generated code, calls mdlOutputs, which then calls my_alg.

```

wrapper.c
...
mdlOutputs(...)
{
    ...
    my_alg();
}
    
```

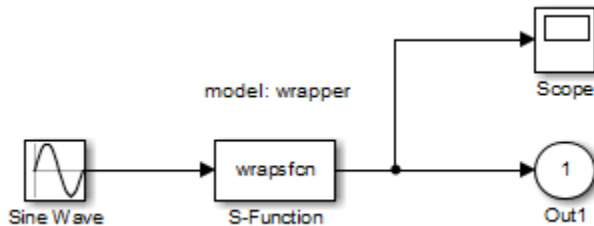
In the TLC wrapper version of the S-function, mdlOutputs in wrapper.exe calls my_alg.

*See note below

*The dotted line is the path taken if the S-function does not have a TLC wrapper file. If there is no TLC wrapper file, the generated code calls mdlOutputs.

Using an S-function wrapper to import algorithms in your Simulink model means that the S-function serves as an interface that calls your C/C++ algorithms from mdlOutputs. S-function wrappers have the advantage that you can quickly integrate large standalone C/C++ programs into your model without having to make changes to the code.

The following sample model includes an S-function wrapper.



There are two files associated with the `wrapsfcn` block, the S-function wrapper and the C/C++ code that contains the algorithm. The S-function wrapper code for `wrapsfcn.c` appears below. The first three statements do the following:

- 1 Defines the name of the S-function (what you enter in the Simulink S-Function block dialog).
- 2 Specifies that the S-function is using the level 2 format.
- 3 Provides access to the `SimStruct` data structure, which contains pointers to data used during simulation and code generation and defines macros that store data in and retrieve data from the `SimStruct`.

For more information, see “Templates for C S-Functions” in the Simulink documentation.

```
#define S_FUNCTION_NAME wrapsfcn
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"

extern real_T my_alg(real_T u); /* Declare my_alg as extern */

/*
 * mdlInitializeSizes - initialize the sizes array
 */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams( S, 0); /*number of input arguments*/

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 1);
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    if (!ssSetNumOutputPorts(S,1)) return;
    ssSetOutputPortWidth(S, 0, 1);

    ssSetNumSampleTimes( S, 1);
}
```

```

/*
 * mdlInitializeSampleTimes - indicate that this S-function runs
 * at the rate of the source (driving block)
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

/*
 * mdlOutputs - compute the outputs by calling my_alg, which
 * resides in another module, my_alg.c
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    real_T *y = ssGetOutputPortRealSignal(S,0);
    *y = my_alg(*uPtrs[0]); /* Call my_alg in mdlOutputs */
}
/*
 * mdlTerminate - called when the simulation is terminated.
 */
static void mdlTerminate(SimStruct *S)
{
}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif

```

The S-function routine `mdlOutputs` contains a function call to `my_alg`, which is the C function containing the algorithm that the S-function performs. This is the code for `my_alg.c`:

```

#ifdef MATLAB_MEX_FILE
#include "tmwtypes.h"
#else
#include "rtwtypes.h"
#endif
real_T my_alg(real_T u)
{
    return(u * 2.0);
}

```

For more information, see the topic “Dependencies When Interfacing Custom and Generated Code” on page 19-39.

The wrapper S-function `wrapsfcn` calls `my_alg`, which computes `u * 2.0`. To build `wrapsfcn.mex`, use the following command:

```
mex wrapsfcn.c my_alg.c
```

TLC S-Function Wrapper

This topic describes how to inline the call to `my_alg` in the `mdlOutputs` section of the generated code. In the above example, the call to `my_alg` is embedded in the `mdlOutputs` section as

```
*y = my_alg(*uPtrs[0]);
```

When you are creating a TLC S-function wrapper, the goal is to embed the same type of call in the generated code.

It is instructive to look at how the code generator executes S-functions that are not inlined. A noninlined S-function is identified by the absence of the file `sfunction.tlc` and the existence of `sfunction.mex`. When generating code for a noninlined S-function, the code generator produces a call to `mdlOutputs` through a function pointer that, in this example, then calls `my_alg`.

The wrapper example contains one S-function, `wrapsfcn.mex`. You must compile and link an additional module, `my_alg`, with the generated code. To do this, specify

```
set_param('wrapper/S-Function', 'SFunctionModules', 'my_alg')
```

Code Overhead for Noninlined S-Functions

The code generated when using `grt.tlc` as the system target file *without* `wrapsfcn.tlc` is

<Generated code comments for wrapper model with **noninlined** wrapsfcn S-function>

```
#include <math.h>
#include <string.h>
#include "wrapper.h"
#include "wrapper.prm"

/* Start the model */
void mdlStart(void)
{
    /* (start code not required) */
}

/* Compute block outputs */
void mdlOutputs(int_T tid)
{
    /* Sin Block: <Root>/Sin */
    rtB.Sin = rtP.Sin.Amplitude *
        sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);
```

```

/* Level2 S-Function Block: <Root>/S-Function (wrapsfcn) */
{
  /* Noninlined S-functions create a SimStruct object and
   * generate a call to S-function routine mdlOutputs
   */
  SimStruct *rts = ssGetSFunction(rtS, 0);
  sfcnOutputs(rts, tid);
}

/* Outport Block: <Root>/Out */
rtY.Out = rtB.S_Function;
}

/* Perform model update */
void mdlUpdate(int_T tid)
{
  /* (update code not required) */
}

/* Terminate function */
void mdlTerminate(void)
{
  /* Level2 S-Function Block: <Root>/S-Function (wrapsfcn) */
  {
    /* Noninlined S-functions require a SimStruct object and
     * the call to S-function routine mdlTerminate
     */
    SimStruct *rts = ssGetSFunction(rtS, *0);
    sfcnTerminate(rts);
  }
}

#include "wrapper.reg"

/* [EOF] wrapper.c */

```

In addition to the overhead outlined above, the `wrapper.reg` generated file contains the initialization of the `SimStruct` for the wrapper S-Function block. There is one child `SimStruct` for each S-Function block in your model. You can significantly reduce this overhead by creating a TLC wrapper for the S-function.

How to Inline

The generated code makes the call to your S-function, `wrapsfcn.c`, in `mdlOutputs` by using this code:

```

SimStruct *rts = ssGetSFunction(rtS, 0);
sfcnOutputs(rts, tid);

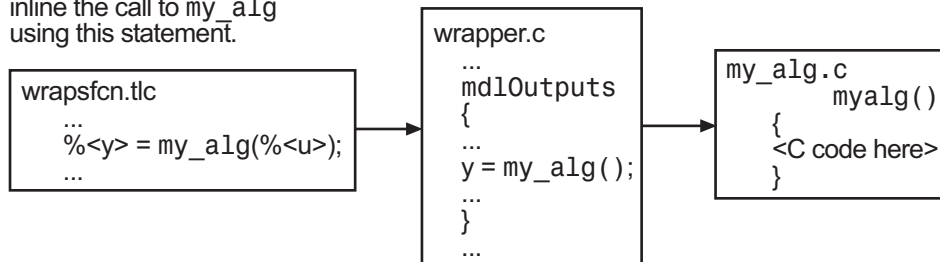
```

This call has computational overhead associated with it. First, the Simulink engine creates a `SimStruct` data structure for the S-Function block. Second, the code generator

constructs a call through a function pointer to execute `mdlOutputs`, then `mdlOutputs` calls `my_alg`. By inlining the call to your C/C++ algorithm, `my_alg`, you can eliminate both the `SimStruct` and the extra function call, thereby improving the efficiency and reducing the size of the generated code.

Inlining a wrapper S-function requires an `sfunction.tlc` file for the S-function (see the “Target Language Compiler” for details). The TLC file must contain the function call to `my_alg`. The following figure shows the relationships between the algorithm, the wrapper S-function, and the `sfunction.tlc` file.

The `wrapsfcn.tlc` file tells Simulink Coder how to inline the call to `my_alg` using this statement.



To inline this call, you have to place your function call in an `sfunction.tlc` file with the same name as the S-function (in this example, `wrapsfcn.tlc`). This causes the Target Language Compiler to override the default method of placing calls to your S-function in the generated code.

This is the `wrapsfcn.tlc` file that inlines `wrapsfcn.c`.

```

%% File      : wrapsfcn.tlc
%% Abstract:
%%      Example inlined tlc file for S-function wrapsfcn.c
%%

%implements "wrapsfcn" "C"

%% Function: BlockTypeSetup =====
%% Abstract:
%%      Create function prototype in model.h as:
%%      "extern real_T my_alg(real_T u);"
%%
%%function BlockTypeSetup(block, system) void
    %openfile buffer
        extern real_T my_alg(real_T u); /* This line is placed in wrapper.h */

```

```

    %closefile buffer
    %<LibCacheFunctionPrototype(buffer)>
%endfunction %% BlockTypeSetup

%% Function: Outputs =====
%% Abstract:
%%     y = my_alg( u );
%%
%function Outputs(block, system) Output
/* %<Type> Block: %<Name> */
%assign u = LibBlockInputSignal(0, "", "", 0)
%assign y = LibBlockOutputSignal(0, "", "", 0)
%% PROVIDE THE CALLING STATEMENT FOR "algorithm"
%% The following line is expanded and placed in mdlOutputs within wrapper.c
%<y> = my_alg(%<u>);

%endfunction %% Outputs

```

The first section of this code inlines the `wrapsfcn` S-Function block and generates the code in C:

```
%implements "wrapsfcn" "C"
```

The next task is to tell the code generator that the routine `my_alg` needs to be declared external in the generated `wrapper.h` file for any `wrapsfcn` S-Function blocks in the model. You only need to do this once for all `wrapsfcn` S-Function blocks, so use the `BlockTypeSetup` function. In this function, you tell the Target Language Compiler to create a buffer and cache the `my_alg` as `extern` in the `wrapper.h` generated header file.

The final step is the inlining of the call to the function `my_alg`. This is done by the `Outputs` function. In this function, you access the block's input and output and place a direct call to `my_alg`. The call is embedded in `wrapper.c`.

The Inlined Code

The code generated when you inline your wrapper S-function is similar to the default generated code. The `mdlTerminate` function does not contain a call to an empty function and the `mdlOutputs` function now directly calls `my_alg`.

```

void mdlOutputs(int_T tid)
{
    /* Sin Block: <Root>/Sin */
    rtB.Sin = rtP.Sin.Amplitude *
        sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);

    /* S-Function Block: <Root>/S-Function */
    rtB.S_Function = my_alg(rtB.Sin); /* Inlined call to my_alg */
}

```

```

/* Output Block: <Root>/Out */
rtY.Out = rtB.S_Function;
}

```

In addition, `wrapper.reg` does not create a child `SimStruct` for the S-function because the generated code is calling `my_alg` directly. This eliminates over 1 KB of memory usage.

Write Fully Inlined S-Functions

Using the example from “Write Wrapper S-Function and TLC Files” on page 24-106, you could eliminate the call to `my_alg` entirely by specifying the explicit code (that is, `2.0 * u`) in `wrapsfcn.tlc`. This is referred to as a *fully inlined S-function*. While this can improve performance, if you are working with a large amount of C/C++ code, this can be a lengthy task. In addition, you now have to maintain your algorithm in two places, the C/C++ S-function itself and the corresponding TLC file. However, the performance gains might outweigh the disadvantages. To inline the algorithm used in this example, in the `Outputs` section of your `wrapsfcn.tlc` file, instead of writing

```

%<y> = my_alg(%<u>);

use

%<y> = 2.0 * %<u>;

```

This is the code produced in `mdlOutputs`:

```

void mdlOutputs(int_T tid)
{
/* Sin Block: <Root>/Sin */
rtB.Sin = rtP.Sin.Amplitude *
    sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);

/* S-Function Block: <Root>/S-Function */
rtB.S_Function = 2.0 * rtB.Sin; /* Explicit embedding of algorithm */

/* Output Block: <Root>/Out */
rtY.Out = rtB.S_Function;
}

```

The Target Language Compiler has replaced the call to `my_alg` with the algorithm itself.

Multiport S-Function

A more advanced multiport inlined S-function example is `sfun_multiport.c` and `sfun_multiport.tlc`. This S-function illustrates how to create a fully inlined TLC

file for an S-function that contains multiple ports. You might find that looking at this example helps you to understand fully inlined TLC files.

Write Fully Inlined S-Functions with mdlRTW Routine

You can inline more complex S-functions that use the S-function `mdlRTW` routine. The purpose of the `mdlRTW` routine is to provide the code generation process with more information about how the S-function is to be inlined, by creating a parameter record of a nontunable parameter for use with a TLC file. The `mdlRTW` routine does this by placing information in the `model.rtw` file. The `mdlRTW` function is described in the text file `matlabroot/simulink/src/sfuntmpl_doc.c`.

As an example of how to use the `mdlRTW` function, this topic discusses the steps you must take to create a direct-index lookup S-function. Lookup tables are collections of ordered data points of a function. Typically, these tables use some interpolation scheme to approximate values of the associated function between known data points. To incorporate the example lookup table algorithm into a Simulink model, the first step is to write an S-function that executes the algorithm in `mdlOutputs`. To produce the most efficient code, the next step is to create a corresponding TLC file to eliminate computational overhead and improve the speed of the lookup computations.

For your convenience, the Simulink product provides support for two general-purpose lookup 1-D and 2-D algorithms. You can use these algorithms as they are or create a custom lookup table S-function to fit your requirements. This topic illustrates how to create a 1-D lookup S-function, `sfun_directlook.c`, and its corresponding inlined `sfun_directlook.tlc` file (see “Target Language Compiler” for more details). This 1-D direct-index lookup table example illustrates the following concepts that you need to know to create your own custom lookup tables:

- Error checking of S-function parameters
- Caching of information for the S-function that doesn't change during model execution
- How to use the `mdlRTW` function to customize the code generator to produce the optimal code for a given set of block parameters
- How to generate an inlined TLC file for an S-function in a combination of the fully inlined form and/or the wrapper form
- “S-Function RTWdata” on page 24-117
- “Direct-Index Lookup Table Algorithm” on page 24-117

- “Direct-Index Lookup Table Example” on page 24-119

S-Function RTWdata

There is a property of blocks called `RTWdata`, which can be used by the Target Language Compiler when inlining an S-function. `RTWdata` is a structure of character vectors that you can attach to a block. It is saved with the model and placed in the `model.rtw` file when generating code. For example, this set of MATLAB commands,

```
mydata.field1 = 'information for field1';
mydata.field2 = 'information for field2';
set_param(gcf,'RTWdata',mydata)
get_param(gcf,'RTWdata')
```

produces this result:

```
ans =

    field1: 'information for field1'
    field2: 'information for field2'
```

Inside the `model.rtw` file for the associated S-Function block is this information.

```
Block {
  Type                "S-Function"
  RTWdata {
    field1             "information for field1"
    field2             "information for field2"
  }
}
```

Note: `RTWdata` is saved in the model file for S-functions that are not linked to a library. However, `RTWdata` is **not persistent** for S-Function blocks that are linked to a library.

Direct-Index Lookup Table Algorithm

The 1-D lookup table block provided in the Simulink library uses interpolation or extrapolation when computing outputs. This extra accuracy might not be required. In this example, you create a lookup table that directly indexes the output vector (y -data vector) based on the current input (x -data) point.

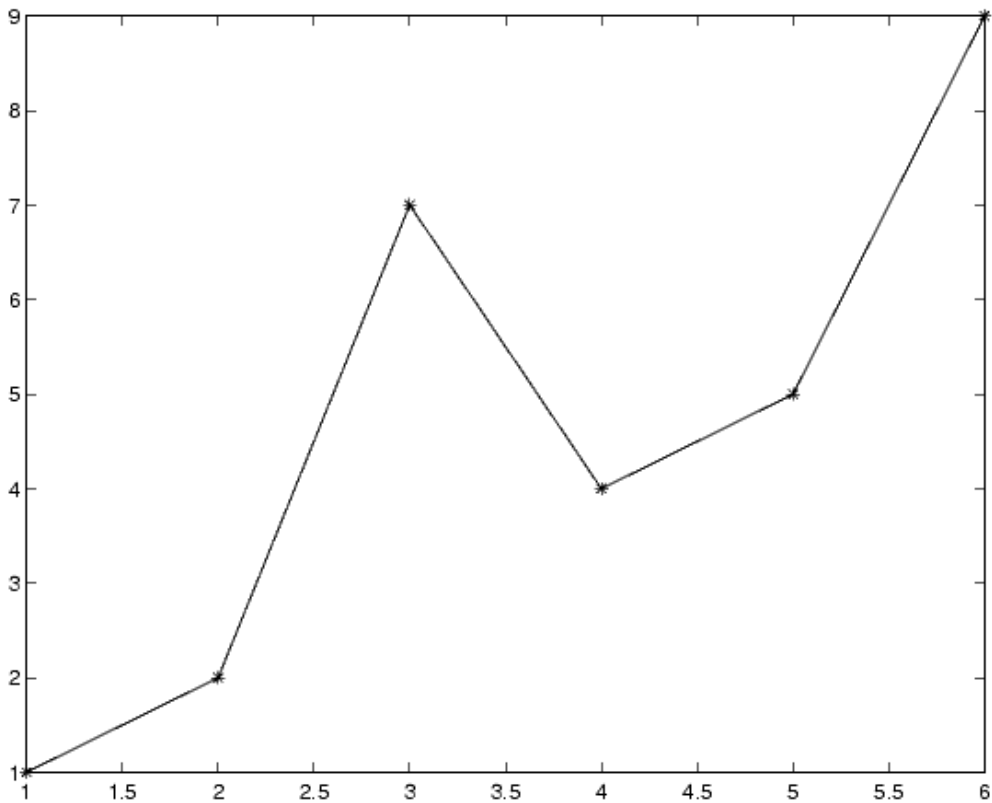
This direct 1-D lookup example computes an approximate solution $p(x)$ to a partially known function $f(x)$ at $x=x_0$, given data point pairs (x,y) in the form of an x -data vector and a y -data vector. For a given data pair (for example, the i 'th pair), $y_i = f(x_i)$. It is assumed that the x -data values are monotonically increasing. If x_0 is outside the range of the x -data vector, the first or last point is returned.

The parameters to the S-function are

XData, YData, XEvenlySpaced

XData and YData are double vectors of equal length representing the values of the unknown function. XDataEvenlySpaced is a scalar, 0.0 for false and 1.0 for true. If the XData vector is evenly spaced, XDataEvenlySpaced is 1.0 and more efficient code is generated.

The following graph shows how the parameters XData=[1:6] and YData=[1,2,7,4,5,9] are handled. For example, if the input (x -value) to the S-Function block is 3, the output (y -value) is 7.



Direct-Index Lookup Table Example

This topic shows how to improve the lookup table by inlining a direct-index S-function with a TLC file. This direct-index lookup table S-function does not require a TLC file. Here the example uses a TLC file for the direct-index lookup table S-function to reduce the code size and increase efficiency of the generated code.

Implementation of the direct-index algorithm with inlined TLC file requires the S-function main module, `sfun_directlook.c`, and a corresponding `lookup_index.c` module. The `lookup_index.c` module contains the `GetDirectLookupIndex` function that is used to locate the index in the `XData` for the current `x` input value when the `XData` is unevenly spaced. The `GetDirectLookupIndex` routine is called from both the S-function and the generated code. Here the example uses the wrapper concept for sharing C/C++ code between Simulink MEX-files and the generated code.

If the `XData` is evenly spaced, then both the S-function main module and the generated code contain the lookup algorithm (not a call to the algorithm) to compute the `y`-value of a given `x`-value, because the algorithm is short. This illustrates the use of a fully inlined S-function for generating optimal code.

The inlined TLC file, which either performs a wrapper call or embeds the optimal C/C++ code, is `sfun_directlook.tlc` (see the example in “mdlRTW Usage” on page 24-120).

Error Handling

In this example, the `mdlCheckParameters` routine verifies that

- The new parameter settings are valid.
- `XData` and `YData` are vectors of the same length containing real finite numbers.
- `XDataEvenlySpaced` is a scalar.
- The `XData` vector is a monotonically increasing vector and evenly spaced.

The `mdlInitializeSizes` function explicitly calls `mdlCheckParameters` after it verifies the number of parameters passed to the S-function. After the Simulink engine calls `mdlInitializeSizes`, it then calls `mdlCheckParameters` whenever you change the parameters or there is a need to reevaluate them.

User Data Caching

The `mdlStart` routine shows how to cache information that does not change during the simulation (or while the generated code is executing). The example caches the value

of the `XDataEvenlySpaced` parameter in `UserData`, a field of the `SimStruct`. The following line in `mdlInitializeSizes` tells the Simulink engine to disallow changes to `XDataEvenlySpaced`.

```
ssSetSFcnParamTunable(S, iParam, SS_PRM_NOT_TUNABLE);
```

During execution, `mdlOutputs` accesses the value of `XDataEvenlySpaced` from `UserData` rather than calling the `mxGetPr` MATLAB API function.

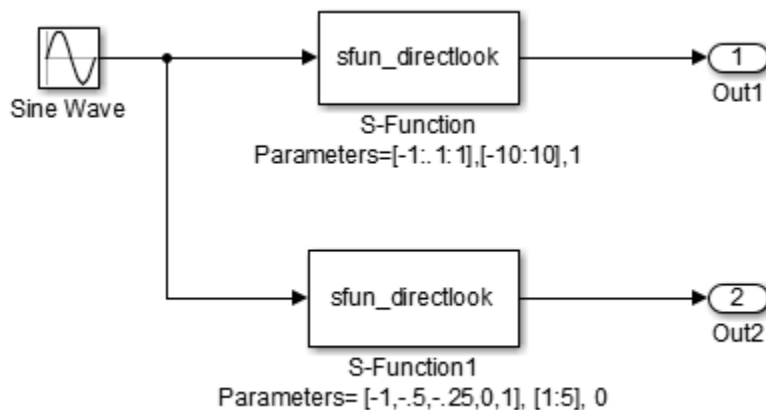
mdlRTW Usage

The code generator calls the `mdlRTW` routine while generating the `model.rtw` file. To produce optimal code for your Simulink model, you can add information to the `model.rtw` file about the mode in which your S-Function block is operating.

The following example adds parameter settings to the `model.rtw` file. The parameter settings do not change during execution. In this case, the `XDataEvenlySpaced` S-function parameter cannot change during execution (`ssSetSFcnParamTunable` was specified as false (0) for it in `mdlInitializeSizes`). The example writes it out as a parameter setting (`XSpacing`) using the function `ssWriterTWParamSettings`.

Because `xData` and `yData` are registered as run-time parameters in `mdlSetWorkWidths`, the code generator handles writing to the `model.rtw` file automatically.

Before examining the S-function and the inlined TLC file, consider the generated code for the following model.



The model uses evenly spaced XData in the top S-Function block and unevenly spaced XData in the bottom S-Function block. When creating this model, you need to specify the following for each S-Function block.

```
set_param('sfun_directlook_ex/S-Function','SFunctionModules','lookup_index')
set_param('sfun_directlook_ex/S-Function1','SFunctionModules','lookup_index')
```

This informs the build process to use the module `lookup_index.c` when creating the executable.

When generating code for this model, the code generator uses the S-function `mdlRTW` method to generate a `model.rtw` file with the value `EvenlySpaced` for the `XSpacing` parameter for the top S-Function block, and the value `UnEvenlySpaced` for the `XSpacing` parameter for the bottom S-Function block. The TLC-file uses the value of `XSpacing` to determine what algorithm to include in the generated code. The generated code contains the lookup algorithm when the XData is evenly spaced, but calls the `GetDirectLookupIndex` routine when the XData is unevenly spaced. The generated `model.c` or `model.cpp` code for the lookup table example model is similar to the following:

```
/*
 * sfun_directlook_ex.c
 *
 * Code generation for Simulink model
 * "sfun_directlook_ex.slx".
 *
...
 */

#include "sfun_directlook_ex.h"
#include "sfun_directlook_ex_private.h"

/* External outputs (root outputs fed by signals with auto storage) */
ExtY_sfun_directlook_ex_T sfun_directlook_ex_Y;

/* Real-time model */
RT_MODEL_sfun_directlook_ex_T sfun_directlook_ex_M;
RT_MODEL_sfun_directlook_ex_T *const sfun_directlook_ex_M =
    &sfun_directlook_ex_M;

/* Model output function */
void sfun_directlook_ex_output(void)
{
    /* local block i/o variables */
    real_T rtb_SFunction;
    real_T rtb_SFunction1;

    /* Sin: '<Root>/Sine Wave' */
    rtb_SFunction1 = sin(sfun_directlook_ex_M->Timing.t[0]);

/* Code that is inlined for the top S-function block in the
```

```

* sfun_directlook_ex model
*/
/* S-Function (sfun_directlook): '<Root>/S-Function' */
{
    const real_T *xData = sfun_directlook_ex_ConstP.SFunction_XData;
    const real_T *yData = sfun_directlook_ex_ConstP.SFunction_YData;
    real_T spacing = xData[1] - xData[0];
    if (rtb_SFunction1 <= xData[0] ) {
        rtb_SFunction = yData[0];
    } else if (rtb_SFunction1 >= yData[20] ) {
        rtb_SFunction = yData[20];
    } else {
        int_T idx = (int_T)( ( rtb_SFunction1 - xData[0] ) / spacing );
        rtb_SFunction = yData[idx];
    }
}

/* Outport: '<Root>/Out1' */
sfun_directlook_ex_Y.Out1 = rtb_SFunction;

/* Code that is inlined for the bottom S-function block in the
* sfun_directlook_ex model
*/
/* S-Function (sfun_directlook): '<Root>/S-Function1' */
{
    const real_T *xData = sfun_directlook_ex_ConstP.SFunction1_XData;
    const real_T *yData = sfun_directlook_ex_ConstP.SFunction1_YData;
    int_T idx;
    idx = GetDirectLookupIndex(xData, 5, rtb_SFunction1);
    rtb_SFunction1 = yData[idx];
}

/* Outport: '<Root>/Out2' */
sfun_directlook_ex_Y.Out2 = rtb_SFunction1;
}

/* Model update function */
void sfun_directlook_ex_update(void)
{
    /* signal main to stop simulation */
    {
        /* Sample time: [0.0s, 0.0s] */
        if ((rtmGetTFinal(sfun_directlook_ex_M)!=-1) &&
            !((rtmGetTFinal(sfun_directlook_ex_M)-sfun_directlook_ex_M->Timing.t[0])
              > sfun_directlook_ex_M->Timing.t[0] * (DBL_EPSILON))) {
            rtmSetErrorStatus(sfun_directlook_ex_M, "Simulation finished");
        }
    }

    /* Update absolute time for base rate */
    /* The "clockTick0" counts the number of times the code of this task has
    * been executed. The absolute time is the multiplication of "clockTick0"
    * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
    * overflow during the application lifespan selected.
    * Timer of this task consists of two 32 bit unsigned integers.
    * The two integers represent the low bits Timing.clockTick0 and the high bits

```

```

    * Timing.clockTickH0. When the low bit overflows to 0, the high bits increment.
    */
    if (!(++sfun_directlook_ex_M->Timing.clockTick0)) {
        ++sfun_directlook_ex_M->Timing.clockTickH0;
    }

    sfun_directlook_ex_M->Timing.t[0] = sfun_directlook_ex_M->Timing.clockTick0 *
        sfun_directlook_ex_M->Timing.stepSize0 +
        sfun_directlook_ex_M->Timing.clockTickH0 *
        sfun_directlook_ex_M->Timing.stepSize0 * 4294967296.0;
}
...

```

matlabroot/toolbox/simulink/simdemos/simfeatures/src/sfun_directlook.c

```

/*
* File      : sfun_directlook.c
* Abstract:
*
*   Direct 1-D lookup. Here we are trying to compute an approximate
*   solution, p(x) to an unknown function f(x) at x=x0, given data point
*   pairs (x,y) in the form of a x data vector and a y data vector. For a
*   given data pair (say the i'th pair), we have y_i = f(x_i). It is
*   assumed that the x data values are monotonically increasing. If the
*   x0 is outside of the range of the x data vector, then the first or
*   last point will be returned.
*
*   This function returns the "nearest" y0 point for a given x0.
*   Interpolation is not performed.
*
*   The S-function parameters are:
*       XData          - double vector
*       YData          - double vector
*       XDataEvenlySpacing - double scalar 0 (false) or 1 (true)
*       The third parameter cannot be changed during simulation.
*
*   To build:
*       mex sfun_directlook.c lookup_index.c
*
* Copyright 1990-2012 The MathWorks, Inc.
*/

#define S_FUNCTION_NAME  sfun_directlook
#define S_FUNCTION_LEVEL 2

#include <math.h>
#include "simstruc.h"
#include <float.h>

/* use utility function IsRealVect() */
#ifdef MATLAB_MEX_FILE
#include "sfun_slutils.h"
#endif

/*=====

```

```

* Build checking *
*=====*/
#if !defined(MATLAB_MEX_FILE)
/*
* This file cannot be used directly with Simulink Coder. However,
* this S-function does work with Simulink Coder via
* the Target Language Compiler technology. See matlabroot/
* toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_directlook.tlc
* for the C version
*/
# error This_file_can_be_used_only_during_simulation_inside_Simulink
#endif

/*=====*
* Defines *
*=====*/

#define XVECT_PIDX          0
#define YVECT_PIDX          1
#define XDATAEVENLYSPACED_PIDX 2
#define NUM_PARAMS          3

#define XVECT(S)            ssGetSFcnParam(S,XVECT_PIDX)
#define YVECT(S)            ssGetSFcnParam(S,YVECT_PIDX)
#define XDATAEVENLYSPACED(S) ssGetSFcnParam(S,XDATAEVENLYSPACED_PIDX)

/*=====*
* misc defines *
*=====*/
#if !defined(TRUE)
#define TRUE 1
#endif
#if !defined(FALSE)
#define FALSE 0
#endif

/*=====*
* typedef's *
*=====*/

typedef struct SFcnCache_tag {
    boolean_T evenlySpaced;
} SFcnCache;

/*=====*
* Prototype define for the function in separate file lookup_index.c *
*=====*/
extern int_T GetDirectLookupIndex(const real_T *x, int_T xlen, real_T u);

/*=====*
* S-function methods *
*=====*/

```

```

#define MDL_CHECK_PARAMETERS          /* Change to #undef to remove function */
#if defined(MDL_CHECK_PARAMETERS) && defined(MATLAB_MEX_FILE)
/* Function: mdlCheckParameters =====
 * Abstract:
 * This routine will be called after mdlInitializeSizes, whenever
 * parameters change or get re-evaluated. The purpose of this routine is
 * to verify the new parameter settings.
 *
 * You should add a call to this routine from mdlInitalizeSizes
 * to check the parameters. After setting your sizes elements, you should:
 *   if (ssGetSFcnParamsCount(S) == ssGetNumSFcnParams(S)) {
 *       mdlCheckParameters(S);
 *   }
 */
static void mdlCheckParameters(SimStruct *S)
{
    if (!IsRealVect(XVECT(S))) {
        ssSetErrorStatus(S,"1st, X-vector parameter must be a real finite "
            " vector");
        return;
    }

    if (!IsRealVect(YVECT(S))) {
        ssSetErrorStatus(S,"2nd, Y-vector parameter must be a real finite "
            "vector");
        return;
    }

    /*
     * Verify that the dimensions of X and Y are the same.
     */
    if (mxGetNumberOfElements(XVECT(S)) != mxGetNumberOfElements(YVECT(S)) ||
        mxGetNumberOfElements(XVECT(S)) == 1) {
        ssSetErrorStatus(S,"X and Y-vectors must be of the same dimension "
            "and have at least two elements");
        return;
    }

    /*
     * Verify we have a valid XDataEvenlySpaced parameter.
     */
    if ((!mxIsNumeric(XDATAEVENLYSPACED(S)) &&
        !mxIsLogical(XDATAEVENLYSPACED(S))) ||
        mxIsComplex(XDATAEVENLYSPACED(S)) ||
        mxGetNumberOfElements(XDATAEVENLYSPACED(S)) != 1) {
        ssSetErrorStatus(S,"3rd, X-evenly-spaced parameter must be logical
scalar");
        return;
    }

    /*
     * Verify x-data is correctly spaced.

```

```

*/
{
    size_t    i;
    boolean_T spacingEqual;
    real_T    *xData = mxGetPr(XVECT(S));
    size_t    numEl = mxGetNumberOfElements(XVECT(S));

    /*
     * spacingEqual is TRUE if user XDataEvenlySpaced
     */
    spacingEqual = (mxGetScalar(XDATAEVENLYSPACED(S)) != 0.0);

    if (spacingEqual) { /* XData is 'evenly-spaced' */
        boolean_T badSpacing = FALSE;
        real_T    spacing = xData[1] - xData[0];
        real_T    space;

        if (spacing <= 0.0) {
            badSpacing = TRUE;
        } else {
            real_T eps = DBL_EPSILON;

            for (i = 2; i < numEl; i++) {
                space = xData[i] - xData[i-1];
                if (space <= 0.0 ||
                    fabs(space-spacing) >= 128.0*eps*spacing ){
                    badSpacing = TRUE;
                    break;
                }
            }
        }

        if (badSpacing) {
            ssSetErrorStatus(S,"X-vector must be an evenly spaced "
                "strictly monotonically increasing vector");
            return;
        }
    } else { /* XData is 'unevenly-spaced' */
        for (i = 1; i < numEl; i++) {
            if (xData[i] <= xData[i-1]) {
                ssSetErrorStatus(S,"X-vector must be a strictly "
                    "monotonically increasing vector");
                return;
            }
        }
    }
}
#endif /* MDL_CHECK_PARAMETERS */

/* Function: mdlInitializeSizes =====
 * Abstract:
 * The sizes information is used by Simulink to determine the S-function

```

```

*   block's characteristics (number of inputs, outputs, states, and so on).
*/
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, NUM_PARAMS); /* Number of expected parameters */

    /*
    * Check parameters passed in, providing the correct number was specified
    * in the S-function dialog box. If an incorrect number of parameters
    * was specified, Simulink will detect the error since ssGetNumSFcnParams
    * and ssGetSFcnParamsCount will differ.
    *   ssGetNumSFcnParams   - This sets the number of parameters your
    *                           S-function expects.
    *   ssGetSFcnParamsCount - This is the number of parameters entered by
    *                           the user in the Simulink S-function dialog box.
    */
#ifdef MATLAB_MEX_FILE
    if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
        mdlCheckParameters(S);
        if (ssGetErrorStatus(S) != NULL) {
            return;
        }
    } else {
        return; /* Parameter mismatch will be reported by Simulink */
    }
#endif

    {
        int iParam = 0;
        int nParam = ssGetNumSFcnParams(S);

        for ( iParam = 0; iParam < nParam; iParam++ )
        {
            switch ( iParam )
            {
                case XDATAEVENLYSPACED_PIDX:

                    ssSetSFcnParamTunable( S, iParam, SS_PRM_NOT_TUNABLE );
                    break;

                default:
                    ssSetSFcnParamTunable( S, iParam, SS_PRM_TUNABLE );
                    break;
            }
        }
    }

    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, DYNAMICALLY_SIZED);
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);

```

```

    ssSetInputPortOverWritable(S, 0, TRUE);

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);

    ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);

    ssSetNumSampleTimes(S, 1);

    ssSetOptions(S,
        SS_OPTION_WORKS_WITH_CODE_REUSE |
        SS_OPTION_EXCEPTION_FREE_CODE |
        SS_OPTION_USE_TLC_WITH_ACCELERATOR);

} /* mdlInitializeSizes */

/* Function: mdlInitializeSampleTimes =====
 * Abstract:
 *   The lookup inherits its sample time from the driving block.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
    ssSetModelReferenceSampleTimeDefaultInheritance(S);
} /* end mdlInitializeSampleTimes */

/* Function: mdlSetWorkWidths =====
 * Abstract:
 *   Set up the [X,Y] data as run-time parameters
 *   that is, these values can be changed during execution.
 */
#define MDL_SET_WORK_WIDTHS
static void mdlSetWorkWidths(SimStruct *S)
{
    const char_T *rtParamNames[] = {"XData", "YData"};
    ssRegAllTunableParamsAsRunTimeParams(S, rtParamNames);
}

#define MDL_START /* Change to #undef to remove function */
#if defined(MDL_START)
/* Function: mdlStart =====
 * Abstract:
 *   Here we cache the state (true/false) of the XDATAEVENLYSPACED parameter.
 *   We do this primarily to illustrate how to "cache" parameter values (or
 *   information which is computed from parameter values) which do not change
 *   for the duration of the simulation (or in the generated code). In this
 *   case, rather than repeated calls to mxGetPr, we save the state once.
 *   This results in a slight increase in performance.
 */
static void mdlStart(SimStruct *S)
{
    SFcnCache *cache = malloc(sizeof(SFcnCache));

```



```

    if (cache == NULL) {
        ssSetErrorStatus(S,"memory allocation error");
        return;
    }

    ssSetUserData(S, cache);

    if (mxGetScalar(XDATAEVENLYSPACED(S)) != 0.0){
        cache->evenlySpaced = TRUE;
    }else{
        cache->evenlySpaced = FALSE;
    }
}
#endif /* MDL_START */

/* Function: mdlOutputs =====
 * Abstract:
 *   In this function, you compute the outputs of your S-function
 *   block. Generally outputs are placed in the output vector, ssGetY(S).
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    SFcnCache      *cache = ssGetUserData(S);
    real_T          *xData = mxGetPr(XVECT(S));
    real_T          *yData = mxGetPr(YVECT(S));
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    real_T          *y      = ssGetOutputPortRealSignal(S,0);
    size_t          ny      = ssGetOutputPortWidth(S,0);
    size_t          xLen    = mxGetNumberOfElements(XVECT(S));
    size_t          i;

    /*
     * When the XData is evenly spaced, we use the direct lookup algorithm
     * to calculate the lookup
     */
    if (cache->evenlySpaced) {
        real_T spacing = xData[1] - xData[0];
        for (i = 0; i < ny; i++) {
            real_T u = *uPtrs[i];

            if (u <= xData[0]) {
                y[i] = yData[0];
            } else if (u >= xData[xLen-1]) {
                y[i] = yData[xLen-1];
            } else {
                int_T idx = (int_T)((u - xData[0])/spacing);
                y[i] = yData[idx];
            }
        }
    }
} else {
    /*

```

```

    * When the XData is unevenly spaced, we use a bisection search to
    * locate the lookup index.
    */
    for (i = 0; i < ny; i++) {
        int_T idx = GetDirectLookupIndex(xData,xLen,*uPtrs[i]);
        y[i] = yData[idx];
    }
}

} /* end mdlOutputs */

/* Function: mdlTerminate =====
 * Abstract:
 *   Free the cache which was allocated in mdlStart.
 */
static void mdlTerminate(SimStruct *S)
{
    SFcnCache *cache = ssGetUserData(S);
    if (cache != NULL) {
        free(cache);
    }
} /* end mdlTerminate */

#define MDL_RTW /* Change to #undef to remove function */
#if defined(MDL_RTW) && (defined(MATLAB_MEX_FILE) || defined(NRT))
/* Function: mdlRTW =====
 * Abstract:
 *   This function is called when Simulink Coder is generating the
 *   model.rtw file. In this routine, you can call the following functions
 *   which add fields to the model.rtw file.
 *
 *   Important! Since this S-function has this mdlRTW method, it is required
 *   to have a corresponding .tlc file so as to work with Simulink Coder. See the
 *   sfun_directlook.tlc in matlabroot/toolbox/simulink/simdemos/simfeatures/tlc_c/.
 */
static void mdlRTW(SimStruct *S)
{
    /*
     * Write out the spacing setting as a param setting, that is, this cannot be
     * changed during execution.
     */
    {
        boolean_T even = (mxGetScalar(XDATAEVENLYSPACED(S)) != 0.0);

        if (!ssWriteRTWParamSettings(S, 1,
                                     SSWRITE_VALUE_QSTR,
                                     "XSpacing",
                                     even ? "EvenlySpaced" : "UnEvenlySpaced")){
            return; /* An error occurred which will be reported by Simulink */
        }
    }
}

```

```

}
#endif /* MDL_RTW */

/*=====
 * Required S-function trailer *
 *=====*/

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif

/* [EOF] sfun_directlook.c */

matlabroot/toolbox/simulink/simdemos/simfeatures/src/lookup_index.c

/* File : lookup_index.c
 * Abstract:
 *
 * Contains a routine used by the S-function sfun_directlookup.c to
 * compute the index in a vector for a given data value.
 *
 * Copyright 1990-2014 The MathWorks, Inc.
 */
#ifdef MATLAB_MEX_FILE
#include <tmwtypes.h>
#else
#include "rtwtypes.h"
#endif

/*
 * Function: GetDirectLookupIndex =====
 * Abstract:
 * Using a bisection search to locate the lookup index when the x-vector
 * isn't evenly spaced.
 *
 * Inputs:
 * *x : Pointer to table, x[0] ...x[xlen-1]
 * xlen : Number of values in xtable
 * u : input value to look up
 *
 * Output:
 * idx : the index into the table such that:
 * if u is negative
 * x[idx] <= u < x[idx+1]
 * else
 * x[idx] < u <= x[idx+1]
 */
int_T GetDirectLookupIndex(const real_T *x, int_T xlen, real_T u)
{
    int_T idx = 0;
    int_T bottom = 0;
    int_T top = xlen-1;

```

```

/*
 * Deal with the extreme cases first:
 *
 * i) u <= x[bottom] then idx = bottom
 * ii) u >= x[top] then idx = top-1
 *
 */
if (u <= x[bottom]) {
    return(bottom);
} else if (u >= x[top]) {
    return(top);
}

/*
 * We have: x[bottom] < u < x[top], onward
 * with search for the index ...
 */
for (;;) {
    idx = (bottom + top)/2;
    if (u < x[idx]) {
        top = idx;
    } else if (u > x[idx+1]) {
        bottom = idx + 1;
    } else {
        /*
         * We have: x[idx] <= u <= x[idx+1], only need
         * to do two more checks and we have the answer
         */
        if (u < 0) {
            /*
             * We want right continuity, that is,
             * if u == x[idx+1]
             * then x[idx+1] <= u < x[idx+2]
             * else x[idx ] <= u < x[idx+1]
             */
            return( (u == x[idx+1]) ? (idx+1) : idx);
        } else {
            /*
             * We want left continuity, that is,
             * if u == x[idx]
             * then x[idx-1] < u <= x[idx ]
             * else x[idx ] < u <= x[idx+1]
             */
            return( (u == x[idx]) ? (idx-1) : idx);
        }
    }
}
} /* end GetDirectLookupIndex */

/* [EOF] lookup_index.c */

matlabroot/toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_directlook.tlc

%% File      : sfun_directlook.tlc
%% Abstract:

```

```

%%      Level-2 S-function sfun_directlook block target file.
%%      It is using direct lookup algorithm without interpolation
%%
%% Copyright 1990-2010 The MathWorks, Inc.
%%

%implements "sfun_directlook" "C"

%% Function: BlockTypeSetup =====
%% Abstract:
%%      Place include and function prototype in the model's header file.
%%
%%function BlockTypeSetup(block, system) void

    %% To add this external function's prototype in the header of the generated
    %% file.
    %%
    %%openfile buffer
    extern int_T GetDirectLookupIndex(const real_T *x, int_T xlen, real_T u);
    %closefile buffer

    %<LibCacheFunctionPrototype(buffer)>

%endfunction

%% Function: mdlOutputs =====
%% Abstract:
%%      Direct 1-D lookup table S-function example.
%%      Here we are trying to compute an approximate solution, p(x) to an
%%      unknown function f(x) at x=x0, given data point pairs (x,y) in the
%%      form of a x data vector and a y data vector. For a given data pair
%%      (say the i'th pair), we have y_i = f(x_i). It is assumed that the x
%%      data values are monotonically increasing. If the first or last x is
%%      outside of the range of the x data vector, then the first or last
%%      point will be returned.
%%
%%      This function returns the "nearest" y0 point for a given x0.
%%      Interpolation is not performed.
%%
%%      The S-function parameters are:
%%      XData
%%      YData
%%      XEvenlySpaced: 0 or 1
%%      The third parameter cannot be changed during execution and is
%%      written to the model.rtw file in XSpacing filed of the SFcnParamSettings
%%      record as "EvenlySpaced" or "UnEvenlySpaced". The first two parameters
%%      can change during execution and show up in the parameter vector.
%%
%%function Outputs(block, system) Output
    /* %<Type> Block: %<Name> */
    {
    %assign rollVars = ["U", "Y"]
    %%
    %% Load XData and YData as local variables

```

```

%%
const real_T *xData = %<LibBlockParameterAddr(XData, "", "", 0)>;
const real_T *yData = %<LibBlockParameterAddr(YData, "", "", 0)>;
%assign xDataLen = SIZE(XData.Value, 1)
%%
%% When the XData is evenly spaced, we use the direct lookup algorithm
%% to locate the lookup index.
%%
%if SFcnParamSettings.XSpacing == "EvenlySpaced"
    real_T spacing = xData[1] - xData[0];

    %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %assign u = LibBlockInputSignal(0, "", lcv, idx)
    %assign y = LibBlockOutputSignal(0, "", lcv, idx)
    if ( %<u> <= xData[0] ) {
        %<y> = yData[0];
    } else if ( %<u> >= yData[%<xDataLen-1>] ) {
        %<y> = yData[%<xDataLen-1>];
    } else {
        int_T idx = (int_T)( ( %<u> - xData[0] ) / spacing );
        %<y> = yData[idx];
    }
    %%
    %% Generate an empty line if we are not rolling,
    %% so that it looks nice in the generated code.
    %%
    %if lcv == ""

    %endif
    %endroll
%else
    %% When the XData is unevenly spaced, we use a bisection search to
    %% locate the lookup index.
    int_T idx;

    %assign xDataAddr = LibBlockParameterAddr(XData, "", "", 0)
    %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %assign u = LibBlockInputSignal(0, "", lcv, idx)
    idx = GetDirectLookupIndex(xData, %<xDataLen>, %<u>);
    %assign y = LibBlockOutputSignal(0, "", lcv, idx)
    %<y> = yData[idx];
    %%
    %% Generate an empty line if we are not rolling,
    %% so that it looks nice in the generated code.
    %%
    %if lcv == ""

    %endif
    %endroll
%endif
}
%endfunction
%% EOF: sfun_directlook.tlc

```

Guidelines for Writing Inlined S-Functions

- Consider using the block property `RTWdata` (see “S-Function `RTWdata`” on page 24-117). This property is a structure of character vectors that you can associate with a block. The code generator saves the structure with the model in the `model.rtw` file and makes the `.rtw` file more readable. For example, suppose you enter the following commands in the MATLAB Command Window:

```
mydata.field1 = 'information for field1';
mydata.field2 = 'information for field2';
set_param(sfun_block, 'RTWdata', mydata);
```

The `.rtw` file that the code generator produces for the block includes the comments specified in the structure `mydata`.

- Consider using the `mdlRTW` function to inline your C MEX S-function in the generated code. This is useful when you want to
 - Rename tunable parameters in the generated code
 - Introduce nontunable parameters into a TLC file

S-Functions That Support Expression Folding

- “About S-Functions that Support Expression Folding” on page 24-135
- “Categories of Output Expressions” on page 24-136
- “Acceptance or Denial of Requests for Input Expressions” on page 24-140
- “Expression Folding in a TLC Block Implementation” on page 24-142

About S-Functions that Support Expression Folding

This topic describes how you can take advantage of expression folding to increase the efficiency of code generated by your own inlined S-Function blocks, by calling macros provided in the S-Function API. This topic assumes that you are familiar with:

- Writing inlined S-functions (see “S-Function Basics”).
- “Target Language Compiler”

The S-Function API lets you specify whether a given S-Function block should nominally accept expressions at a given input port. A block should not always accept expressions. For example, if the address of the signal at the input is used, expressions should not be accepted at that input, because it is not possible to take the address of an expression.

The S-Function API also lets you specify whether an expression can represent the computations associated with a given output port. When you request an expression at a block's input or output port, the Simulink engine determines whether or not it can honor that request, given the block's context. For example, the engine might deny a block's request to output an expression if the destination block does not accept expressions at its input, if the destination block has an update function, or if multiple output destinations exist.

The decision to honor or deny a request to output an expression can also depend on the category of output expression the block uses (see “Categories of Output Expressions” on page 24-136).

The topics that follow explain

- When and how you can request that a block accept expressions at an input port
- When and how you can request that a block generate expressions at an output port
- The conditions under which the Simulink engine will honor or deny such requests

To take advantage of expression folding in your S-functions, you should understand when to request acceptance and generation of expressions for specific blocks. You do not have to understand the algorithm by which the Simulink engine chooses to accept or deny these requests. However, if you want to trace between the model and the generated code, it is helpful to understand some of the more common situations that lead to denial of a request.

Categories of Output Expressions

When you implement a C MEX S-function, you can specify whether the code corresponding to a block's output is to be generated as an expression. If the block generates an expression, you must specify that the expression is *constant*, *trivial*, or *generic*.

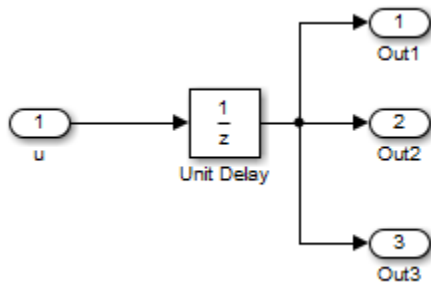
A *constant* output expression is a direct access to one of the block's parameters. For example, the output of a Constant block is defined as a constant expression because the output expression is simply a direct access to the block's `Value` parameter.

A *trivial* output expression is an expression that can be repeated, without a performance penalty, when the output port has multiple output destinations. For example, the output of a Unit Delay block is defined as a trivial expression because the output expression is simply a direct access to the block's state. Because the output expression does not have computations, it can be repeated more than once without degrading the performance of the generated code.

A *generic* output expression is an expression that should be assumed to have a performance penalty if repeated. As such, a generic output expression is not suitable for repeating when the output port has multiple output destinations. For instance, the output of a Sum block is a generic rather than a trivial expression because it is costly to recompute a Sum block output expression as an input to multiple blocks.

Examples of Trivial and Generic Output Expressions

Consider the following block diagram. The Delay block has multiple destinations, yet its output is designated as a trivial output expression, so that it can be used more than once without degrading the efficiency of the code.



The following code excerpt shows code generated from the Unit Delay block in this block diagram. The three root outputs are directly assigned from the state of the Unit Delay block, which is stored in a field of the global data structure `rtDWork`. Since the assignment is direct, without expressions, there is no performance penalty associated with using the trivial expression for multiple destinations.

```
void MdlOutputs(int_T tid)
{
    ...
    /* Output: <Root>/Out1 incorporates:
     *   UnitDelay: <Root>/Unit Delay */
    rtY.Out1 = rtDWork.Unit_Delay_DSTATE;

    /* Output: <Root>/Out2 incorporates:
     *   UnitDelay: <Root>/Unit Delay */
    rtY.Out2 = rtDWork.Unit_Delay_DSTATE;

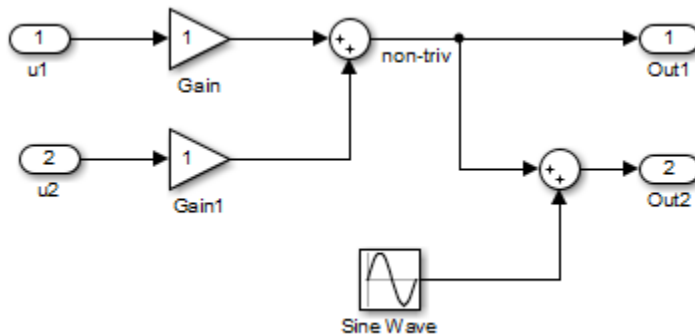
    /* Output: <Root>/Out3 incorporates:
     *   UnitDelay: <Root>/Unit Delay */
    rtY.Out3 = rtDWork.Unit_Delay_DSTATE;
```

```

}
...
}

```

On the other hand, consider the Sum blocks in the following model:



The upper Sum block in the preceding model generates the signal labeled `non_triv`. Computation of this output signal involves two multiplications and an addition. If the Sum block's output were permitted to generate an expression even when the block had multiple destinations, the block's operations would be duplicated in the generated code. In the case illustrated, the generated expressions would proliferate to four multiplications and two additions. This would degrade the efficiency of the program. Accordingly the output of the Sum block is not allowed to be an expression because it has multiple destinations

The code generated for the previous block diagram shows how code is generated for Sum blocks with single and multiple destinations.

The Simulink engine does not permit the output of the upper Sum block to be an expression because the signal `non_triv` is routed to two output destinations. Instead, the result of the multiplication and addition operations is stored in a temporary variable (`rtb_non_triv`) that is referenced twice in the statements that follow, as seen in the code excerpt below.

In contrast, the lower Sum block, which has only a single output destination (`Out2`), does generate an expression.

```

void MdlOutputs(int_T tid)
{
    /* local block i/o variables */
    real_T rtb_non_triv;

```

```

real_T rtb_Sine_Wave;

/* Sum: <Root>/Sum incorporates:
 *   Gain: <Root>/Gain
 *   Inport: <Root>/u1
 *   Gain: <Root>/Gain1
 *   Inport: <Root>/u2
 *
 * Regarding <Root>/Gain:
 *   Gain value: rtP.Gain_Gain
 *
 * Regarding <Root>/Gain1:
 *   Gain value: rtP.Gain1_Gain
 */
rtb_non_triv = (rtP.Gain_Gain * rtU.u1) + (rtP.Gain1_Gain *
rtU.u2);

/* Outport: <Root>/Out1 */
rtY.Out1 = rtb_non_triv;

/* Sin Block: <Root>/Sine Wave */

rtb_Sine_Wave = rtP.Sine_Wave_Amp *
sin(rtP.Sine_Wave_Freq * rtmGetT(rtM_model) +
rtP.Sine_Wave_Phase) + rtP.Sine_Wave_Bias;

/* Outport: <Root>/Out2 incorporates:
 *   Sum: <Root>/Sum1
 */
rtY.Out2 = (rtb_non_triv + rtb_Sine_Wave);
}

```

Specify the Category of an Output Expression

The S-Function API provides macros that let you declare whether an output of a block should be an expression, and if so, to specify the category of the expression. The following table specifies when to declare a block output to be a constant, trivial, or generic output expression.

Types of Output Expressions

Category of Expression	When to Use
Constant	Use only if block output is a direct memory access to a block parameter.
Trivial	Use only if block output is an expression that can appear multiple times in the code without reducing efficiency (for example, a direct memory access to a field of the DWORK vector, or a literal).
Generic	Use if output is an expression, but not constant or trivial.

You must declare outputs as expressions in the `mdlSetWorkWidths` function using macros defined in the S-Function API. The macros have the following arguments:

- `SimStruct *S`: pointer to the block's `SimStruct`.
- `int idx`: zero-based index of the output port.
- `bool value`: pass in `TRUE` if the port generates output expressions.

The following macros are available for setting an output to be a constant, trivial, or generic expression:

- `void ssSetOutputPortConstOutputExprInRTW(SimStruct *S, int idx, bool value)`
- `void ssSetOutputPortTrivialOutputExprInRTW(SimStruct *S, int idx, bool value)`
- `void ssSetOutputPortOutputExprInRTW(SimStruct *S, int idx, bool value)`

The following macros are available for querying the status set by prior calls to the macros above:

- `bool ssGetOutputPortConstOutputExprInRTW(SimStruct *S, int idx)`
- `bool ssGetOutputPortTrivialOutputExprInRTW(SimStruct *S, int idx)`
- `bool ssGetOutputPortOutputExprInRTW(SimStruct *S, int idx)`

The set of generic expressions is a superset of the set of trivial expressions, and the set of trivial expressions is a superset of the set of constant expressions.

Therefore, when you query an output that has been set to be a constant expression with `ssGetOutputPortTrivialOutputExprInRTW`, it returns `True`. A constant expression is considered a trivial expression, because it is a direct memory access that can be repeated without degrading the efficiency of the generated code.

Similarly, an output that has been configured to be a constant or trivial expression returns `True` when queried for its status as a generic expression.

Acceptance or Denial of Requests for Input Expressions

A block can request that its output be represented in code as an expression. Such a request can be denied if the destination block cannot accept expressions at its input port. Furthermore, conditions independent of the requesting block and its destination blocks can prevent acceptance of expressions.

This topic discusses block-specific conditions under which requests for input expressions are denied. For information on other conditions that prevent acceptance of expressions, see “Denial of Block Requests to Output Expressions” on page 24-142.

A block should not be configured to accept expressions at its input port under the following conditions:

- The block must take the address of its input data. It is not possible to take the address of most types of input expressions.
- The code generated for the block references the input more than once (for example, the Abs or Max blocks). This would lead to duplication of a potentially complex expression and a subsequent degradation of code efficiency.

If a block refuses to accept expressions at an input port, then a block that is connected to that input port is not permitted to output a generic or trivial expression.

A request to output a constant expression is not denied, because there is no performance penalty for a constant expression, and the software can take the parameter’s address.

S-Function API to Specify Input Expression Acceptance

The S-Function API provides macros that let you:

- Specify whether a block input should accept nonconstant expressions (that is, trivial or generic expressions)
- Query whether a block input accepts nonconstant expressions

By default, block inputs do not accept nonconstant expressions.

You should call the macros in your `mdlSetWorkWidths` function. The macros have the following arguments:

- `SimStruct *S`: pointer to the block's `SimStruct`.
- `int idx`: zero-based index of the input port.
- `bool value`: pass in `TRUE` if the port accepts input expressions; otherwise pass in `FALSE`.

The macro available for specifying whether or not a block input should accept a nonconstant expression is as follows:

```
void ssSetInputPortAcceptExprInRTW(SimStruct *S, int portIdx, bool value)
```

The corresponding macro available for querying the status set by any prior calls to `ssSetInputPortAcceptExprInRTW` is as follows:

```
bool ssGetInputPortAcceptExprInRTW(SimStruct *S, int portIdx)
```

Denial of Block Requests to Output Expressions

Even after a specific block requests that it be allowed to generate an output expression, that request can be denied for generic reasons. These reasons include, but are not limited to

- The output expression is nontrivial, and the output has multiple destinations.
- The output expression is nonconstant, and the output is connected to at least one destination that does not accept expressions at its input port.
- The output is a test point.
- The output has been assigned an external storage class.
- The output must be stored using global data (for example is an input to a merge block or a block with states).
- The output signal is complex.

You do not need to consider these generic factors when deciding whether or not to utilize expression folding for a particular block. However, these rules can be helpful when you are examining generated code and analyzing cases where the expression folding optimization is suppressed.

Expression Folding in a TLC Block Implementation

To take advantage of expression folding, you must modify the TLC block implementation of an inlined S-Function such that it informs the Simulink engine whether it generates or accepts expressions at its

- Input ports, as explained in “S-Function API to Specify Input Expression Acceptance” on page 24-141.
- Output ports, as explained in “Categories of Output Expressions” on page 24-136.

This topic discusses required modifications to the TLC implementation.

Expression Folding Compliance

In the `BlockInstanceSetup` function of your S-function, register your block to be compliant with expression folding. Otherwise, expression folding requested or allowed at the block's outputs or inputs will be disabled, and temporary variables will be used.

To register expression folding compliance, call the TLC library function `LibBlockSetIsExpressionCompliant(block)`, which is defined in `matlabroot/rtw/c/tlc/lib/utllib.tlc`. For example:

```
%% Function: BlockInstanceSetup =====
%%
%function BlockInstanceSetup(block, system) void
    %%
    %<LibBlockSetIsExpressionCompliant(block)>
    %%
%endfunction
```

You can conditionally disable expression folding at the inputs and outputs of a block by making the call to this function conditionally.

If you override one of the TLC block implementations provided by the code generator with your own implementation, you should not make the preceding call until you have updated your implementation, as described by the guidelines for expression folding in the following topics.

Output Expressions

The `BlockOutputSignal` function is used to generate code for a scalar output expression or one element of a nonscalar output expression. If your block outputs an expression, you should add a `BlockOutputSignal` function. The prototype of the `BlockOutputSignal` is

```
%function BlockOutputSignal(block,system,portIdx,ucv,lcx,idx,retType) void
```

The arguments to `BlockOutputSignal` are as follows:

- **block**: the record for the block for which an output expression is being generated
- **system**: the record for the system containing the block
- **portIdx**: zero-based index of the output port for which an expression is being generated
- **ucv**: user control variable defining the output element for which code is being generated
- **lcx**: loop control variable defining the output element for which code is being generated
- **idx**: signal index defining the output element for which code is being generated
- **retType**: character vector defining the type of signal access desired:
 - "Signal" specifies the contents or address of the output signal.

"SignalAddr" specifies the address of the output signal

The `BlockOutputSignal` function returns a character vector for the output signal or address. The character vector should enforce the precedence of the expression by using opening and terminating parentheses, unless the expression consists of a function call. The address of an expression can only be returned for a constant expression; it is the address of the parameter whose memory is being accessed. The code implementing the `BlockOutputSignal` function for the Constant block is shown below.

```
%% Function: BlockOutputSignal =====
%% Abstract:
%%     Return the reference to the parameter. This function *may*
%%     be used by Simulink when optimizing the Block IO data structure.
%%
%%function BlockOutputSignal(block,system,portIdx,ucv,lcx,idx,retType) void
%switch retType
%case "Signal"
%return LibBlockParameter(Value,ucv,lcx,idx)
%case "SignalAddr"
%return LibBlockParameterAddr(Value,ucv,lcx,idx)
%default
%assign errTxt = "Unsupported return type: %<retType>"
%<LibBlockReportError(block,errTxt)>
%endswitch
%endfunction
```

The code implementing the `BlockOutputSignal` function for the Relational Operator block is shown below.

```
%% Function: BlockOutputSignal =====
%% Abstract:
%%     Return an output expression. This function *may*
%%     be used by Simulink when optimizing the Block IO data structure.
%%
%%function BlockOutputSignal(block,system,portIdx,ucv,lcx,idx,retType) void
%switch retType
%case "Signal"
%assign logicOperator = ParamSettings.Operator
%if ISEQUAL(logicOperator, "~=")
%assign op = "!="
elseif ISEQUAL(logicOperator, "==") %assign op = "=="
%else
%assign op = logicOperator
%endif
%assign u0 = LibBlockInputSignal(0, ucv, lcx, idx)
%assign u1 = LibBlockInputSignal(1, ucv, lcx, idx)
%return "(%<u0> %<op> %<u1>)"
%default
%assign errTxt = "Unsupported return type: %<retType>"
%<LibBlockReportError(block,errTxt)>
%endswitch
```



```
%endfunction
```

Expression Folding for Blocks with Multiple Outputs

When a block has a single output, the `Outputs` function in the block's TLC file is called only if the output port is not an expression. Otherwise, the `BlockOutputSignal` function is called.

If a block has multiple outputs, the `Outputs` function is called if any output port is not an expression. The `Outputs` function should guard against generating code for output ports that are expressions. This is achieved by guarding sections of code corresponding to individual output ports with calls to `LibBlockOutputSignalIsExpr()`.

For example, consider an S-Function with two inputs and two outputs, where

- The first output, `y0`, is equal to two times the first input.
- The second output, `y1`, is equal to four times the second input.

The `Outputs` and `BlockOutputSignal` functions for the S-function are shown in the following code excerpt.

```
%% Function: BlockOutputSignal =====
%% Abstract:
%%     Return an output expression. This function *may*
%%     be used by Simulink when optimizing the Block IO data structure.
%%
%function BlockOutputSignal(block,system,portIdx,ucv,lcx,idx,retType) void
%switch retType
%case "Signal"
    %assign u = LibBlockInputSignal(portIdx, ucv, lcx, idx)
    %case "Signal"
    %if portIdx == 0
        %return "(2 * %<u>)"
    %elseif portIdx == 1
        %return "(4 * %<u>)"
    %endif
%default
%assign errTxt = "Unsupported return type: %<retType>"
%<LibBlockReportError(block,errTxt)>
%endswitch
%endfunction
%%
%% Function: Outputs =====
%% Abstract:
%%     Compute output signals of block
%%
%function Outputs(block,system) Output
%assign rollVars = ["U", "Y"]
%roll sigIdx = RollRegions, lcx = RollThreshold, block, "Roller", rollVars
%assign u0 = LibBlockInputSignal(0, "", lcx, sigIdx)
    %assign u1 = LibBlockInputSignal(1, "", lcx, sigIdx)
```

```

%assign y0 = LibBlockOutputSignal(0, "", lcv, sigIdx)
%assign y1 = LibBlockOutputSignal(1, "", lcv, sigIdx)
%if !LibBlockOutputSignalIsExpr(0)
%<y0> = 2 * %<u0>;
%endif
%if !LibBlockOutputSignalIsExpr(1)
%<y1> = 4 * %<u1>;
%endif
%endroll
%endfunction

```

Comments for Blocks That Are Expression-Folding-Compliant

In the past, blocks preceded their outputs code with comments of the form

```
/* %<Type> Block: %<Name> */
```

When a block is expression-folding-compliant, the initial line shown above is generated automatically. You should not include the comment as part of the block's TLC implementation. Additional information should be registered using the `LibCacheBlockComment` function.

The `LibCacheBlockComment` function takes a character vector as an input, defining the body of the comment, except for the opening header, the final newline of a single or multiline comment, and the closing trailer.

The following TLC code illustrates registering a block comment. Note the use of the function `LibBlockParameterForComment`, which returns a character vector, suitable for a block comment, specifying the value of the block parameter.

```

%openfile commentBuf
$c(*) Gain value: %<LibBlockParameterForComment(Gain)>
%closefile commentBuf
%<LibCacheBlockComment(block, commentBuf)>

```

S-Functions That Specify Port Scope and Reusability

You can use the following `SimStruct` macros in the `mdlInitializeSizes` method to specify the scope and reusability of the memory used for your S-function's input and output ports:

- `ssSetInputPortOptimOpts`: Specify the scope and reusability of the memory allocated to an S-function input port
- `ssSetOutputPortOptimOpts`: Specify the scope and reusability of the memory allocated to an S-function output port
- `ssSetInputPortOverWritable`: Specify whether one of your S-function's input ports can be overwritten by one of its output ports

- `ssSetOutputPortOverwritesInputPort`: Specify whether an output port can share its memory buffer with an input port

You declare an input or output as local or global, and indicate its reusability, by passing one of the following four options to the `ssSetInputPortOptimOpts` and `ssSetOutputPortOptimOpts` macros:

- `SS_NOT_REUSABLE_AND_GLOBAL`: Indicates that the input and output ports are stored in separate memory locations in the global block input and output structure
- `SS_NOT_REUSABLE_AND_LOCAL`: Indicates that the code generator can declare individual local variables for the input and output ports
- `SS_REUSABLE_AND_LOCAL`: Indicates that the code generator can reuse a single local variable for these input and output ports
- `SS_REUSABLE_AND_GLOBAL`: Indicates that these input and output ports are stored in a single element in the global block input and output structure

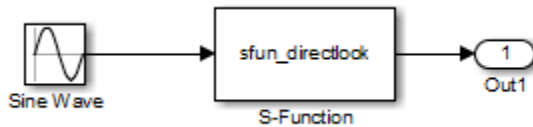
Note Marking an input or output port as a local variable does not imply that the code generator uses a local variable in the generated code. If your S-function accesses the inputs and outputs only in its `mdlOutputs` routine, the code generator declares the inputs and outputs as local variables. However, if the inputs and outputs are used elsewhere in the S-function, the code generator includes them in the global block input and output structure.

The reusability setting indicates if the memory associated with an input or output port can be overwritten. To reuse input and output port memory:

- 1 Indicate the ports are reusable using either the `SS_REUSABLE_AND_LOCAL` or `SS_REUSABLE_AND_GLOBAL` option in the `ssSetInputPortOptimOpts` and `ssSetOutputPortOptimOpts` macros
- 2 Indicate the input port memory is overwritable using `ssSetInputPortOverWritable`
- 3 If your S-function has multiple input and output ports, use `ssSetOutputPortOverwritesInputPort` to indicate which output and input ports share memory

The following example shows how different scope and reusability settings affect the generated code. The following model contains an S-function block pointing to the C

MEX S-function `matlabroot/toolbox/simulink/simdemos/simfeatures/src/sfun_directlook.c`, which models a direct 1-D lookup table.



The S-function's `mdlInitializeSizes` method declares the input port as reusable, local, and overwritable and the output port as reusable and local, as follows:

```
static void mdlInitializeSizes(SimStruct *S)
{
/* snip */
    ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);
    ssSetInputPortOverWritable(S, 0, TRUE);

/* snip */
    ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);

/* snip */
}
```

The generated code for this model stores the input and output signals in a single local variable `rtb_SFunction`, as shown in the following output function:

```
static void sl_directlook_output(int_T tid)
{
/* local block i/o variables */
    real_T rtb_SFunction[2];

/* Sin: '<Root>/Sine Wave' */
    rtb_SFunction[0] = sin(((real_T)sl_directlook_DWork.counter[0] +
        sl_directlook_P.SineWave_Offset) * 2.0 * 3.1415926535897931E+000 /
        sl_directlook_P.SineWave_NumSamp) * sl_directlook_P.SineWave_Amp[0] +
        sl_directlook_P.SineWave_Bias;
    rtb_SFunction[1] = sin(((real_T)sl_directlook_DWork.counter[1] +
        sl_directlook_P.SineWave_Offset) * 2.0 * 3.1415926535897931E+000 /
        sl_directlook_P.SineWave_NumSamp) * sl_directlook_P.SineWave_Amp[1] +
        sl_directlook_P.SineWave_Bias;

/* S-Function Block: <Root>/S-Function */
{
    const real_T *xData = &sl_directlook_P.SFunction_XData[0];
    const real_T *yData = &sl_directlook_P.SFunction_YData [0];
    real_T spacing = xData[1] - xData[0];
    if (rtb_SFunction[0] <= xData[0] ) {
        rtb_SFunction[0] = yData[0];
    } else if (rtb_SFunction[0] >= yData[20] ) {
```

```

    rtb_SFunction[0] = yData[20];
} else {
    int_T idx = (int_T)( ( rtb_SFunction[0] - xData[0] ) / spacing );
    rtb_SFunction[0] = yData[idx];
}

if (rtb_SFunction[1] <= xData[0] ) {
    rtb_SFunction[1] = yData[0];
} else if (rtb_SFunction[1] >= yData[20] ) {
    rtb_SFunction[1] = yData[20];
} else {
    int_T idx = (int_T)( ( rtb_SFunction[1] - xData[0] ) / spacing );
    rtb_SFunction[1] = yData[idx];
}
}

/* Outport: '<Root>/Out1' */
sl_directlook_Y.Out1[0] = rtb_SFunction[0];
sl_directlook_Y.Out1[1] = rtb_SFunction[1];
UNUSED_PARAMETER(tid);
}

```

The following table shows variations of the code generated for this model when using the generic real-time target (GRT). Each row explains a different setting for the scope and reusability of the S-function's input and output ports.

Scope and reusability	S-function mdlInitializeSizes code	Generated code
Inputs: Local, reusable, overwriteable Outputs: Local, reusable	<pre> ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL); ssSetInputPortOverWritable(S, 0, TRUE); ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL); </pre>	<p>The <i>model.c</i> file declares a local variable in the output function.</p> <pre> /* local block i/o variables */ real_T rtb_SFunction[2]; </pre>
Inputs: Global, reusable, overwriteable Outputs: Global, reusable	<pre> ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_GLOBAL); ssSetInputPortOverWritable(S, 0, TRUE); ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_GLOBAL); </pre>	<p>The <i>model.h</i> file defines a block signals structure with a single element to store the S-function's input and output.</p> <pre> /* Block signals (auto storage) */ typedef struct { real_T SFunction[2]; } BlockIO_sl_directlook; </pre> <p>The <i>model.c</i> file uses this element of the structure in calculations of the S-function's input and output signals.</p> <pre> /* Sin: '<Root>/Sine Wave' */ </pre>

Scope and reusability	S-function mdlInitializeSizes code	Generated code
		<pre>sl_directlook_B.SFunction[0] = sin ... /* snip */ /*S-Function Block:<Root>/S-Function*/ { const real_T *xData = &sl_directlook_P.SFunction_XData[0]</pre>
<p>Inputs: Local, not reusable</p> <p>Outputs: Local, not reusable</p>	<pre>ssSetInputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_LOCAL); ssSetInputPortOverWritable(S, 0, FALSE); ssSetOutputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_LOCAL);</pre>	<p>The <i>model.c</i> file declares local variables for the S-function's input and output in the output function</p> <pre>/* local block i/o variables */ real_T rtb_SineWave[2]; real_T rtb_SFunction[2];</pre>
<p>Inputs: Global, not reusable</p> <p>Outputs: Global, not reusable</p>	<pre>ssSetInputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_GLOBAL); ssSetInputPortOverWritable(S, 0, FALSE); ssSetOutputPortOptimOpts(S, 0, SS_NOT_REUSABLE_AND_GLOBAL);</pre>	<p>The <i>model.h</i> file defines a block signal structure with individual elements to store the S-function's input and output.</p> <pre>/* Block signals (auto storage) */ typedef struct { real_T SineWave[2]; real_T SFunction[2]; } BlockIO_sl_directlook;</pre> <p>The <i>model.c</i> file uses the different elements in this structure when calculating the S-function's input and output.</p> <pre>/* Sin: '<Root>/Sine Wave' */ sl_directlook_B.SineWave[0] = sin ... /* snip */ /*S-Function Block:<Root>/S-Function*/ { const real_T *xData = &sl_directlook_P.SFunction_XData[0]</pre>

S-Functions That Specify Sample Time Inheritance Rules

For the Simulink engine to determine whether a model can inherit a sample time, the S-functions in the model need to specify how they use sample times. You can specify this information by calling the macro `ssSetModelReferenceSampleTimeInheritanceRule` from `mdlInitializeSizes` or `mdlSetWorkWidths`. To use this macro:

- 1 Check whether the S-function calls any of the following macros:

- `ssGetSampleTime`
- `ssGetInputPortSampleTime`
- `ssGetOutputPortSampleTime`
- `ssGetInputPortOffsetTime`
- `ssGetOutputPortOffsetTime`
- `ssGetSampleTimePtr`
- `ssGetInputPortSampleTimeIndex`
- `ssGetOutputPortSampleTimeIndex`
- `ssGetSampleTimeTaskID`
- `ssGetSampleTimeTaskIDPtr`

2 Check for the following in your S-function TLC code:

- `LibBlockSampleTime`
- `CompiledModel.SampleTime`
- `LibBlockInputSignalSampleTime`
- `LibBlockInputSignalOffsetTime`
- `LibBlockOutputSignalSampleTime`
- `LibBlockOutputSignalOffsetTime`

3 Depending on your search results, use `ssSetModelReferenceSampleTimeInheritanceRule` as indicated in the following table.

If...	Use...
None of the macros or functions are present, the S-function does not preclude the model from inheriting a sample time.	<code>ssSetModelReferenceSampleTimeInheritanceRule(S, USE_DEFAULT_FOR_DISCRETE_INHERITANCE)</code>
Any of the macros or functions are used for <ul style="list-style-type: none"> • Throwing errors if sample time is inherited, continuous, or constant 	<code>ssSetModelReferenceSampleTimeInheritanceRule... (S,USE_DEFAULT_FOR_DISCRETE_INHERITANCE)</code>

If...	Use...
<ul style="list-style-type: none"> • Checking <code>ssIsSampleHit</code> • Checking whether sample time is inherited in either <code>mdlSetInputPortSampleTime</code> or <code>mdlSetOutputPortSampleTime</code> before setting 	
The S-function uses its sample time for computing parameters, outputs, and so on	<code>ssSetModelReferenceSampleTimeInheritanceRule(S, DISALLOW_SAMPLE_TIME_INHERITANCE)</code>

Note If an S-function does not set the `ssSetModelReferenceSampleTimeInheritanceRule` macro, by default the Simulink engine assumes that the S-function does not preclude the model containing that S-function from inheriting a sample time. However, the engine issues a warning indicating that the model includes S-functions for which this macro is not set.

You can use settings on the **All Parameters** pane of the Configuration Parameters dialog box or Model Explorer to control how the Simulink engine responds when it encounters S-functions that have unspecified sample time inheritance rules. Toggle the **Unspecified inheritability of sample time** diagnostic to none, warning, or error. The default is warning.

S-Functions That Support Code Reuse

You can reuse the generated code for identical subsystems that occur in multiple instances within a model and across referenced models. For more information about code generation of subsystems for code reuse, see “Code Generation of Subsystems” on page 6-2. If you want your S-function to support code reuse for a subsystem, the S-function must meet the following requirements:

- The S-function must be inlined.
- Code generated from the S-function must not use static variables.
- The S-function must initialize its pointer work vector in `mdlStart` and not before.
- The S-function must not be a sink that logs data to the workspace.

- The S-function must register its parameters as run-time parameters in `mdlSetWorkWidths`. (It must not use `ssWriteRTWParameters` in its `mdlRTW` function for this purpose.)
- The S-function must not be a device driver.

In addition to meeting the preceding requirements, your S-function must set the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag (see the description of `ssSetOptions` in the Simulink Writing S-Function documentation). This flag indicates that your S-function meets the requirements for subsystem code reuse.

S-Functions for Multirate Multitasking Environments

- “About S-Functions for Multirate Multitasking Environments” on page 24-153
- “Rate Grouping Support in S-Functions” on page 24-153
- “Create Multitasking, Multirate, Port-Based Sample Time S-Functions” on page 24-154

About S-Functions for Multirate Multitasking Environments

S-functions can be used in models with multiple sample rates and deployed in multitasking target environments. Likewise, S-functions themselves can have multiple rates at which they operate. The code generator produces code for multirate multitasking models using an approach called *rate grouping*. In code generated for ERT-based targets, rate grouping generates separate `model_step` functions for the base rate task and each subrate task in the model. Although rate grouping is a code generation feature found in ERT targets only, your S-functions can use it in other contexts when you code them as explained below.

Rate Grouping Support in S-Functions

To take advantage of rate grouping, you must inline your multirate S-functions if you have not done so. You need to follow certain Target Language Compiler protocols to exploit rate grouping. Coding TLC to exploit rate grouping does not prevent your inlined S-functions from functioning properly in GRT. Likewise, your inlined S-functions will still generate valid ERT code even if you do not make them rate-grouping-compliant. If you do so, however, they will generate more efficient code for multirate models.

For instructions and examples of Target Language Compiler code illustrating how to create and upgrade S-functions to generate rate-grouping-compliant code, see “Rate Grouping Compliance and Compatibility Issues” in the Embedded Coder documentation.

For each multirate S-function that is not rate grouping-compliant, the code generator issues the following warning when you build:

```
Warning: Simulink Coder: Code of output function for multirate block
'<Root>/S-Function' is guarded by sample hit checks rather than being rate
grouped. This will generate the same code for all rates used by the block,
possibly generating dead code. To avoid dead code, you must update the TLC
file for the block.
```

You will also find a comment such as the following in code generated for each noncompliant S-function:

```
/* Because the output function of multirate block
   <Root>/S-Function is not rate grouped,
   the following code might contain unreachable blocks of code.
   To avoid this, you must update your block TLC file. */
```

The words “update function” are substituted for “output function” in these warnings.

Create Multitasking, Multirate, Port-Based Sample Time S-Functions

The following instructions show how to support both data determinism and data integrity in multirate S-functions. They do not cover cases where there is no determinism nor integrity. Support for frame-based processing does not affect the requirements.

Note The slow rates must be multiples of the fastest rate. The instructions do not apply when two rates being interfaced are not multiples or when the rates are not periodic.

Rules for Properly Handling Fast-to-Slow Transitions

The rules that multirate S-functions should observe for inputs are

- The input should only be read at the rate that is associated with the input port sample time.
- Generally, the input data is written to DWork, and the DWork can then be accessed at the slower (downstream) rate.

The input can be read at every sample hit of the input rate and written into DWork memory, but this DWork memory cannot then be directly accessed by the slower rate. DWork memory that will be read by the slow rate must only be written by the fast rate when there is a *special sample hit*. A special sample hit occurs when both this input port rate and rate to which it is interfacing have a hit. Depending on their requirements and design, algorithms can process the data in several locations.

The rules that multirate S-functions should observe for outputs are

- The output should not be written by a rate other than the rate assigned to the output port, except in the optimized case described below.
- The output should always be written when the sample rate of the output port has a hit.

If these conditions are met, the S-Function block can specify that the input port and output port can both be made local and reusable.

You can include an optimization when little or no processing needs to be done on the data. In such cases, the input rate code can directly write to the output (instead of by using `DWork`) when there is a special sample hit. If you do this, however, you must declare the output port to be *global* and *not reusable*. This optimization results in one less `memcpy` but does introduce nonuniform processing requirements on the faster rate.

Whether you use this optimization or not, the most recent input data, as seen by the slower rate, is the value when both the faster and slower rate had their hits (and possible earlier input data as well, depending on the algorithm). Subsequent steps by the faster rate and the associated input data updates are not seen by the slower rate until the next hit for the slow rate occurs.

Pseudocode Examples of Fast-to-Slow Rate Transition

The pseudocode below abstracts how you should write your C MEX code to handle fast-to-slow transitions, illustrating with an input rate of 0.1 second driving an output rate of one second. A similar approach can be taken when inlining the code. The block has following characteristics:

- File: `sfun_multirate_zoh.c`, Equation: $y = u(\text{tslow})$
- Input: local and reusable
- Output: local and reusable
- DirectFeedthrough: yes

```
OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSpecialSampleHit("1")) {
        DWork = u;
    }
}
if (ssIsSampleHit("1")) {
    y = DWork;
}
```

An alternative, slightly optimized approach for simple algorithms:

- Input: local and reusable
- Output: global and not reusable because it needs to persist between special sample hits
- DirectFeedthrough: yes

```
OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSpecialSampleHit("1")) {
        y = u;
    }
}
```

Example adding a simple algorithm:

- File: `sfun_multirate_avg.c`; Equation: $y = \text{average}(u)$
- Input: local and reusable
- Output: local and reusable
- DirectFeedthrough: yes

(Assume `DWork[0:10]` and `DWork[mycounter]` are initialized to zero)

```
OutputFcn
if (ssIsSampleHit(".1")) {
    /* In general, processing on 'u' could be done here,
       it runs on every hit of the fast rate. */
    DWork[DWork[mycounter]++] = u;
    if (ssIsSpecialSampleHit("1")) {
        /* In general, processing on DWork[0:10] can be done
           here, but it does cause the faster rate to have
           nonuniform processing requirements (every 10th hit,
           more code needs to be run).*/
        DWork[10] = sum(DWork[0:9])/10;
        DWork[mycounter] = 0;
    }
}
if (ssIsSampleHit("1")) {
    /* Processing on DWork[10] can be done here before
       outputting. This code runs on every hit of the
       slower task. */
    y = DWork[10];
}
```

Rules for Properly Handling Slow-to-Fast Transitions

When output rates are faster than input rates, input should only be read at the rate that is associated with the input port sample time, observing the following rules:

- Always read input from the update function.
- Use no special sample hit checks when reading input.
- Write the input to a DWork.
- When there is a special sample hit between the rates, copy the DWork into a second DWork in the output function.
- Write the second DWork to the output at every hit of the output sample rate.

The block can request that the input port be made local but it cannot be set to reusable. The output port can be set to local and reusable.

As in the fast-to-slow transition case, the input should not be read by a rate other than the one assigned to the input port. Similarly, the output should not be written to at a rate other than the rate assigned to the output port.

An optimization can be made when the algorithm being implemented is only required to run at the slow rate. In such cases, you use only one DWork. The input still writes to the DWork in the update function. When there is a special sample hit between the rates, the output function copies the same DWork directly to the output. You must set the output port to be global and not reusable in this case. This optimization results in one less `memcpy` operation per special sample hit.

In either case, the data that the fast rate computations operate on is always delayed, that is, the data is from the previous step of the slow rate code.

Pseudocode Examples of Slow-to-Fast Rate Transition

The pseudocode below abstracts what your S-function needs to do to handle slow-to-fast transitions, illustrating with an input rate of one second driving an output rate of 0.1 second. The block has following characteristics:

- File: `sfun_multirate_delay.c`, Equation: $y = u(ts_{\text{slow}} - 1)$
- Input: Set to local, will be local if output/update are combined (ERT) otherwise will be global. Set to not reusable because input needs to be preserved until the update function runs.
- Output: local and reusable
- DirectFeedthrough: no

```
OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSpecialSampleHit("1")) {
        DWork[1] = DWork[0];
    }
}
```

```

    }
    y = DWork[1];
}
UpdateFcn
if (ssIsSampleHit("1")) {
    DWork[0] = u;
}

```

An alternative, optimized approach can be used by some algorithms:

- Input: Set to local, will be local if output/update are combined (ERT) otherwise will be global. Set to not reusable because input needs to be preserved until the update function runs.
- Output: global and not reusable because it needs to persist between special sample hits.
- DirectFeedthrough: no

```

OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSpecialSampleHit("1")) {
        y = DWork;
    }
}
UpdateFcn
if (ssIsSampleHit("1")) {
    DWork = u;
}

```

Example adding a simple algorithm:

- File: `sfun_multirate_modulate.c`, Equation: $y = \sin(t_{\text{fast}}) + u(t_{\text{slow}} - 1)$
- Input: Set to local, will be local if output/update are combined (an ERT feature) otherwise will be global. Set to not reusable because input needs to be preserved until the update function runs.
- Output: local and reusable
- DirectFeedthrough: no

```

OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSpecialSampleHit("1")) {
        /* Processing not likely to be done here. It causes
        * the faster rate to have nonuniform processing
        * requirements (every 10th hit, more code needs to
        * be run).*/
        DWork[1] = DWork[0];
    }
    /* Processing done at fast rate */
    y = sin(ssGetTaskTime(".1")) + DWork[1];
}

```

```
}
UpdateFcn
if (ssIsSampleHit("1")) {
    /* Processing on 'u' can be done here. There is a delay of
       one slow rate period before the fast rate sees it.*/
    DWork[0] = u;}
}
```

See Also

legacy_code

Related Examples

- “Introduction to the Target Language Compiler”
- “Inlining S-Functions”
- “Build S-Function With Legacy Code Tool” on page 24-27

Build Support for S-Functions

About Build Support for S-Functions

User-written S-Function blocks provide a powerful way to incorporate external code into the Simulink development environment. In most cases, you use S-functions to integrate existing external code with generated code. Several approaches to writing S-functions are available:

- “Write Noninlined S-Function and TLC Files” on page 24-104
- “Write Wrapper S-Function and TLC Files” on page 24-106
- “Write Fully Inlined S-Functions” on page 24-115
- “Write Fully Inlined S-Functions with mdlRTW Routine” on page 24-116
- “S-Functions That Support Code Reuse” on page 24-152
- “S-Functions for Multirate Multitasking Environments” on page 24-153

S-functions also provide the most flexible and capable way of including build information for legacy and custom code files in the build process.

There are a number of different ways of adding S-functions to the build process.

Implicit Build Support

When building models with S-functions, the build process adds rules, include paths, and source file names to the generated makefile. The source files (`.h`, `.c`, and `.cpp`) for the S-function must be in the same folder as the S-function MEX-file. Whether using the toolchain approach or template makefile approach for builds, the build process propagates this information through the toolchain or template makefile.

The implicit build support includes:

- If the file `sfcnname.h` exists in the same folder as the S-function MEX-file (for example, `sfcnname.mexext`), the folder is added to the include path.
- If the file `sfcnname.c` or `sfcnname.cpp` exists in the same folder as the S-function MEX-file, the build process adds a makefile rule for compiling files from that folder.
- When an S-function is not inlined with a TLC file, the build process must compile the S-function source file. To determine the name of the source file to add to the list

of files to compile, the build process searches for `sfcname.cpp` on the MATLAB path. If the source file is found, the build process adds the source file name to the makefile. If `sfcname.cpp` is not found on the path, the build process adds the file name `sfcname.c` to the makefile, whether or not it is on the MATLAB path.

Note: For the Simulink engine to find the MEX-file for simulation and code generation, it must exist on the MATLAB path or be in your current MATLAB working folder.

Specify Additional Source Files for an S-Function

If your S-function has additional source file dependencies, you must add the names of the additional modules to the build process. Specify the filenames:

- In the **S-function modules** field of the S-Function block parameter dialog box
- With the `SFunctionModules` parameter in a call to the `set_param` function

For example, suppose you build your S-function with multiple modules.

```
mex sfun_main.c sfun_module1.c sfun_module2.c
```

You can then add the modules to the build process by doing one of the following:

- In the S-function block dialog box, specify `sfun_main`, `sfun_module1`, and `sfun_module2` in the **S-function modules** field.
- At the MATLAB command prompt, enter:

```
set_param(sfun_block, 'SFunctionModules', 'sfun_module1 sfun_module2')
```

Alternatively, you can define a variable to represent the parameter value.

```
modules = 'sfun_module1 sfun_module2'
set_param(sfun_block, 'SFunctionModules', modules)
```

Note: The **S-function modules** field and `SFunctionModules` parameter do not support complete source file path specifications. To use the parameter, the code generator must be able to find the additional source files when executing the makefile. For the code generator to locate the additional files, place them in the same folder as the S-function MEX-file. You can then leverage the implicit build support described in “Implicit Build Support” on page 24-160.

When you are ready to generate code after using the **S-function modules** field or `SFunctionModules` parameter, you must force the code generator to rebuild the top model, as described in “Control Regeneration of Top Model Code” on page 25-43.

For more complicated S-function file dependencies, such as specifying source files in other locations or specifying libraries or object files, use the `rtwmakecfg.m` API, as described in “Use `rtwmakecfg.m` API to Customize Generated Makefiles” on page 35-24.

Use TLC Library Functions

If you inline your S-function by writing a TLC file, you can add source file names to the build process by using the TLC library function `LibAddToModelSources`. For details, see “`LibAddSourceFileCustomSection(file, builtInSection, newSection)`” in the Target Language Compiler documentation.

Note: This function does not support complete source file path specifications and assumes that the code generator can find the additional source files when executing the makefile.

Another useful TLC library function is `LibAddToCommonIncludes`. Use this function in a `#include` statement to include S-function header files in the generated `model.h` header file. For details, see “`LibAddToCommonIncludes(incFileName)`” in the Target Language Compiler documentation.

For more complicated S-function file dependencies, such as specifying source files in other locations or specifying libraries or object files, use the `rtwmakecfg.m` API, as described in “Use `rtwmakecfg.m` API to Customize Generated Makefiles” on page 35-24.

Precompile S-Function Libraries

You can precompile new or updated S-function libraries (MEX-files) for a model by using the MATLAB language function `rtw_precompile_libs`. Using a specified model and a library build specification, this function builds and places the libraries in a precompiled library folder.

By precompiling S-function libraries, you can optimize system builds. Once your precompiled libraries exist, the build process can omit library compilation from

subsequent builds. For models that use numerous libraries, the time savings for build processing can be significant.

To use `rtw_precompile_libs`:

- 1 Set the library file suffix, including the file type extension, based on the platform in use.
- 2 Set the precompiled library folder.
- 3 Define a build specification.
- 4 Issue a call to `rtw_precompile_libs`.

The procedure describes these steps in more detail.

- 1 Set the library file suffix, including the file type extension, based on the platform.

Consider determining the type of platform and then use the `TargetLibSuffix` parameter to set the library suffix accordingly. For example, when applying a suffix for a GRT target, you might set the suffix to `_std.a` for a UNIX platform and `_vcx64.lib` for a Windows platform.

```
if isunix
    suffix = '_std.a';
else
    suffix = '_vcx64.lib';
end
```

```
set_param(my_model, 'TargetLibSuffix', suffix);
```

There are a number of factors that influence the precompiled library suffix and extension. The following table provides examples for typical selections of system target file, the compiler toolchain, and other options that affect your choice of suffix and extension. For more information, examine the template make files in the `matlab/rtw/c/grt` folder or `matlab/rtw/c/ert` folder.

TMF File	COMPILER _TOOL_CHAIN Value	Precompiler Libraries (PRECOMP_LIBRARIES)			
		Library Suffix S-Function (EXPAND _LIBRARY _NAME Value)	Library Suffix Integer- Only Code (EXPAND _LIBRARY _NAME Value)	Library Suffix Optimize for Speed (EXPAND _LIBRARY _NAME Value)	Library Extension (EXPAND _LIBRARY _NAME Value)
ert_lcc64.tmf	lcc	_rtwsfcn_lcc	_int_ert_lcc	_ert_lcc	.lib

TMF File	COMPILER _TOOL_CHAIN Value	Precompiler Libraries (PRECOMP_LIBRARIES)			
		Library Suffix S-Function (EXPAND _LIBRARY _NAME Value)	Library Suffix Integer- Only Code (EXPAND _LIBRARY _NAME Value)	Library Suffix Optimize for Speed (EXPAND _LIBRARY _NAME Value)	Library Extension (EXPAND _LIBRARY _NAME Value)
ert_vcx64.tmf	vcx64	_rtwsfcn_vcx64	_int_ert_vcx64	_ert_vcx64	.lib
ert_unix.tmf	unix	_rtwsfcn	_int_ert	_ert	.a
grt_lcc64.tmf	lcc	n/a	n/a	_lcc	.lib
grt_vcx64.tmf	vcx64	n/a	n/a	_vcx64	.lib
grt_unix.tmf	unix	n/a	n/a	_std	.a

2 Set the precompiled library folder.

Use one of the following methods to set the precompiled library folder.

- Set the `TargetPreCompLibLocation` parameter, as explained in “Specify the Location of Precompiled Libraries” on page 35-9.
- Set the `makeInfo.precompile` field in an `rtwmakecfg.m` function file. (For more information, see “Use `rtwmakecfg.m` API to Customize Generated Makefiles” on page 35-24.)

If you set both `TargetPreCompLibLocation` and `makeInfo.precompile`, the setting for `TargetPreCompLibLocation` takes precedence.

The following command sets the precompiled library folder for model `my_model` to folder `lib` under the current working folder.

```
set_param(my_model, 'TargetPreCompLibLocation', fullfile(pwd, 'lib'));
```

Note: If you set both the target folder for the precompiled library files and a target library file suffix, the build process detects whether any precompiled library files are missing while processing builds.

3 Define a build specification.

Set up a structure that defines a build specification. The following table describes fields that you can define in the structure. These fields are optional, except for `rtwmakecfgDirs`.

Field	Description
<code>rtwmakecfgDirs</code>	<p>A cell array of character vectors that name the folders containing <code>rtwmakecfg</code> files for libraries to be precompiled. The function uses the <code>Name</code> and <code>Location</code> elements of <code>makeInfo.library</code>, as returned by <code>rtwmakecfg</code>, to specify the name and location of the precompiled libraries. If you set the <code>TargetPreCompLibLocation</code> parameter to specify the library folder, that setting overrides the <code>makeInfo.library.Location</code> setting.</p> <p>Note: The specified model must contain blocks that use precompiled libraries specified by the <code>rtwmakecfg</code> files because the TMF-to-makefile conversion generates the library rules only if the libraries are used.</p>
<code>libSuffix</code>	<p>A character vector that specifies the suffix, including the file type extension, to be appended to the name of each library (for example, <code>.a</code> or <code>_vc.lib</code>). The character vector must include a period (<code>.</code>). You must set the suffix with either this field or the <code>TargetLibSuffix</code> parameter. If you specify a suffix with both mechanisms, the <code>TargetLibSuffix</code> setting overrides the setting of this field.</p>
<code>intOnlyBuild</code>	<p>A Boolean flag. When set to true, the flag indicates the libraries are to be optimized such that they are compiled from integer code only. This field applies to ERT targets only.</p>
<code>makeOpts</code>	<p>A character vector that specifies an option to be included in the <code>rtwMake</code> command line.</p>
<code>addLibs</code>	<p>A cell array of structures that specify libraries to be built that are not specified by an <code>rtwmakecfg</code> function. Each structure must be defined with two fields that are character arrays:</p> <ul style="list-style-type: none"> <code>libName</code> — the name of the library without a suffix <code>libLoc</code> — the location for the precompiled library <p>The TMF can specify other libraries and how those libraries are to be built. Use this field if you need to precompile those libraries.</p>

The following commands set up build specification `build_spec`, which indicates that the files to be compiled are in folder `src` under the current working folder.

```
build_spec = [];
build_spec.rtwmakecfgDirs = {fullfile(pwd,'src')};
```

4 Issue a call to `rtw_precompile_libs`.

Issue a call to `rtw_precompile_libs` that specifies the model for which you want to build the precompiled libraries and the build specification. For example:

```
rtw_precompile_libs(my_model, build_spec);
```

See Also

`legacy_code`

Related Examples

- “Introduction to the Target Language Compiler”
- “Inlining S-Functions”
- “Build S-Function With Legacy Code Tool” on page 24-27

Program Building, Interaction, and Debugging in Simulink Coder

- “Change Programming Language” on page 25-2
- “Select and Configure Compiler or IDE” on page 25-3
- “Troubleshoot Compiler Issues” on page 25-9
- “Configure Build Process” on page 25-13
- “Build a Generic Real-Time Program” on page 25-23
- “Initiate the Build Process” on page 25-35
- “Build and Run a Program” on page 25-36
- “How Executable Programs Are Built From Models” on page 25-39
- “Rebuild a Model” on page 25-41
- “Control Regeneration of Top Model Code” on page 25-43
- “Reduce Build Time for Referenced Models” on page 25-45
- “Relocate Code to Another Development Environment” on page 25-51
- “Executable Program Generation” on page 25-63
- “Profile Code Performance” on page 25-66

Change Programming Language

By default, the code generator produces C code.

To change the language setting:

- 1 In the Configuration Parameters dialog box, on the **Code Generation** pane, select C or C++ from the **Language** menu in the **Target selection** section. Alternatively, set the `TargetLang` parameter at the command line.

The code generator produces `.c` or `.cpp` files, depending on your selection, and places the generated files in your build folder.

For more information, see “Language”.

- 2 Check whether you must choose and configure a compiler. If you select C++, you must choose and configure a compiler. For details, see “Select and Configure Compiler or IDE” on page 25-3.
- 3 Check whether the standard math library is configured for your compiler. By default, the code generator uses the ISO/IEC 9899:1999 C (**C99 (ISO)**) library for the C language and the ISO/IEC 14882:2003 C++ (**C++03 (ISO)**) library for the C++ language.

For more information, see “Standard math library”.

More About

- “Select and Configure Compiler or IDE” on page 25-3
- “Troubleshoot Compiler Issues” on page 25-9

Select and Configure Compiler or IDE

Compilers and the Build Process

The build process requires a supported compiler. *Compiler*, in this context, refers to a development environment containing a linker and make utility, in addition to a high-level language compiler. For details on supported compiler versions, see

http://www.mathworks.com/support/compilers/current_release

When creating an executable program, the build process must be able to access a supported compiler. The build process can automatically find a compiler to use based on your default MEX compiler.

The build process also requires the selection of a toolchain or template makefile. The toolchain or template makefile determines which compiler runs, during the make phase of the build, to compile the generated code.

To determine which templates makefiles are available for your compiler and target, see System Target Files Available from the System Target File Browser.

For both generated files and user-supplied files, the file extension, `.c` or `.cpp`, determines whether the build process uses a C or a C++ compiler. If the file extension is `.c`, the build process uses C compiler to compile the file, and the symbols use the C linkage convention. If the file extension is `.cpp`, the build process uses a C++ compiler to compile the file, and the symbols use the C++ linkage specification.

Language Standards Compliance

The code generator produces code that is compliant with the following standards:

Language	Supported Standard
C	ISO/IEC 9899:1990, also known as C89/C90
C++	ISO/IEC 14882:2003

Code that the code generator produces from these sources is ANSI C/C++ compliant:

- Simulink built-in block algorithmic code
- Generated system level code (task ID [TID] checks, management, functions, and so on)

- Code from other blocksets, including the Fixed-Point Designer product, the Communications System Toolbox product, and so on
- Code from other code generators, such as MATLAB functions

Additionally, the code generator can incorporate code from

- Embedded targets (for example, startup code, device driver blocks)
- User-written S-functions or TLC files

Note: Coding standards for these two sources are beyond the control of the code generator, and can be a source for compliance problems, such as code that uses C99 features not supported in the ANSI C, C89/C90 subset.

Programming Language Considerations

The code generator produces C and C++ code. Consider the following as you choose a programming language:

- Whether you need to configure the code generator to use a specific compiler. This is required to generate C++ code on Windows. See “Select and Configure Compiler or IDE” on page 25-3.
- The language configuration setting for the model. See “Change Programming Language” on page 25-2.
- Whether you need to integrate legacy or custom code with generated code. For a summary of integration options, see “About External Code Integration” on page 24-2.
- Whether you need to integrate C and C++ code. If so, see “About External Code Integration” on page 24-2.

Note: You can mix C and C++ code when integrating generated code with custom code. However, you must be aware of the differences between C and default C++ linkage conventions, and add the `extern "C"` linkage specifier where required. For the details of the differing linkage conventions and how to apply `extern "C"`, refer to a C++ programming language reference book.

- “C++ Language Limitations” on page 25-5.

For an example, enter `sfndemo_cppcount` in the MATLAB Command Window. For a Stateflow example, enter `sf_cpp`.

C++ Language Considerations

To use the C++ target language support, you might need to configure the code generator to use a specific compiler. For example, if a supported compiler is not installed on your Microsoft Windows computer, the default compiler is the lcc C compiler shipped with the MATLAB product. This compiler does not support C++. If you do not configure the code generator to use a C++ compiler before you specify C++ for code generation, the software produces an error message.

C++ Language Limitations

- The code generator does not support C++ code generation for the following:
 - Simscape Driveline
 - Simscape Multibody First Generation (Simscape Multibody Second Generation is supported)
 - Simscape Power Systems
 - Embedded Targets in Embedded Coder
 - Simulink Real-Time
- The following Embedded Coder dialog box fields currently do not accept the `.cpp` extension. However, the code generator produces a `.cpp` file if you specify a filename without an extension in these fields, with C++ selected as the target language for your generated code.
 - **Data definition filename** field on the **Data Placement** pane of the Configuration Parameters dialog box
 - **Definition file** field for a data object in Model Explorer. Data objects are objects of the class `Simulink.Signal`, `Simulink.Parameter`, and subclasses.

These restrictions on specifying `.cpp` will be removed in a future release.

Code Generator Optimizes Code by Relying on Undefined Behavior of C Language for Signed Integer Overflows

The code generator reduces memory usage and enhances performance of code that it produces by assuming signed integer C operations wrap on overflow. A signed integer overflow occurs when the result of an arithmetic operation is outside the range of values that the output data type can represent. The C programming language does not define the results of such operations. Some C compilers aggressively optimize signed operations for in-range values at the expense of overflow conditions. Other compilers preserve the

full wrap-on-overflow behavior. For example, the gcc and MinGW compilers provide an option to reliably wrap overflow on signed integer overflows. The generated program image for a model can produce results that differ from model simulation results because the handling of overflows varies, depending on your compiler.

When you generate code, if you use a supported compiler with the default options configured by the code generator, the compiler preserves the full wrap-on-overflow behavior. If you change the compiler options or compile the code in another development environment, it is possible that the compiler does not preserve the full wrap-on-overflow behavior. In this case, the executable program can produce unpredictable results.

If this issue is a concern for your application, consider one or more of the following actions:

- Verify that the compiled code produces expected results.
- If your compiler has an option to force wrapping behavior, turn it on. For example, for the gcc compiler or a compiler based on gcc, such as MinGW, configure the build process to use the compiler option `-fwrapv`.
- Choose a compiler that wraps on integer overflow.
- If you have Embedded Coder installed, develop and apply a custom code replacement library to replace code generated for signed integers. For more information, see “Code Replacement Customization”.

Choose and Configure Compiler on Microsoft Windows

On a Windows computer, you can install supported compilers. MATLAB automatically selects a default compiler, which is used by the build process. Alternatively, you can use the lcc C compiler that is shipped with the MATLAB product.

The compiler that the code generator requires is determined primarily by the specified system target file:

- If you select a toolchain-based system target file such as `grt.tlc` (Generic Real-Time Target), `ert.tlc` (Embedded Coder), or `autosar.tlc` (Embedded Coder for AUTOSAR), the **Build process** subpane displays toolchain parameters for configuring the build process. Use the **Toolchain** parameter to select a compiler and associated tools for your model build. Click on the **All Parameters > Toolchain > Validate** button to validate the selected toolchain.
- If you select a template makefile (TMF) based system target file, such as a desktop or embedded target, `rsim.tlc`, or a Visual C++ Solution File target, the **Build process**

subpane displays template makefile parameters for configuring the build process. The **Template makefile** parameter displays the default TMF file for the selected target. If the target supports compiler-specific template makefiles (for example, Rapid Simulation or S-Function target), you can set **Template makefile** to a compiler-specific TMF, such as `rsim_lcc.tmf`. (See “Available System Target Files” on page 17-3 for valid TMF names.) If the template makefile is not compiler-specific, you can change the default compiler through the following command:

```
mex -setup
```

Choose and Configure Compiler on UNIX

On a UNIX platform, the build process uses the default compiler. The default compiler is GNU `gcc/g++` for GNU or Xcode for Mac. For more information, see supported compilers. The compiler for your model build is indicated in the **Build process** parameters located on the **Code Generation** pane of the Configuration Parameters dialog box. The type of display depends on the **System target file** selected for your model build:

- If you select a toolchain-based system target file, such as `grt.tlc` (Generic Real-Time Target), `ert.tlc` (Embedded Coder), or `autosar.tlc` (Embedded Coder for AUTOSAR), the **Build process** subpane displays toolchain parameters for configuring the build process. You can use the **Toolchain** parameter to examine and validate the compiler and associated tool settings for your model build.
- If you select a template makefile (TMF) based system target file, such as a desktop or embedded target or `rsim.tlc`, the **Build process** subpane displays template makefile parameters for configuring the build process. The **Template makefile** parameter displays the default TMF file for the selected target. If the target supports compiler-specific template makefiles (for example, Rapid Simulation or S-Function target), set **Template makefile** to the UNIX TMF for your target. For example, for the Rapid Simulation target, enter `rsim_unix.tmf`. (See “Available System Target Files” on page 17-3 for valid UNIX TMF names.) You can display the UNIX compiler with the following command:

```
mex -setup
```

Include S-Function Source Code

When the code generator builds models with S-functions, source code for the S-functions can be either in the current folder or in the same folder as their MEX-file. The code

generator adds an include path to the generated makefiles whenever it finds a file named *sfcnname*.h in the same folder that the S-function MEX-file is in. This folder must be on the MATLAB path.

Similarly, the code generator adds a rule for the folder when it finds a file *sfcnname*.c (or .cpp) in the same folder as the S-function MEX-file is in.

More About

- “Target Environment Configuration”
- “Available System Target Files” on page 17-3
- System Target Files Available from the System Target File Browser
- “Change Programming Language” on page 25-2
- “Troubleshoot Compiler Issues” on page 25-9

External Websites

- http://www.mathworks.com/support/compilers/current_release

Troubleshoot Compiler Issues

Compiler Version Mismatch Errors

Description

The build process produces a compiler version mismatch error.

Action

- 1 Check the list of supported and compatible compilers available at www.mathworks.com/support/compilers/current_release/.
- 2 Upgrade or change your compiler. For more information, see “Choose and Configure Compiler on Microsoft Windows” on page 25-6 or “Choose and Configure Compiler on UNIX” on page 25-7.
- 3 Rebuild the model.

Executable Program Generated for Model Produces Different Results from Model Simulation Results

Description

The generated source code includes an arithmetic operation that produces a signed integer overflow. Your compiler might not implement wrapping behavior for signed integer overflow conditions, or you are using an appropriate compiler, but you did not configure it to use the `-fwrapv` option.

For more information, see “Code Generator Relies on Undefined Behavior of C Language for Integer Overflows.”

Action

- If your compiler has an option to force wrapping behavior, turn it on. For example, for the gcc compiler or a compiler based on gcc, such as MinGW, specify the compiler option `-fwrapv`.
- Choose a compiler that checks for integer overflows.
- If you have an Embedded Coder license, develop and apply a code replacement library to replace code generated for signed integers.

Build Process Generates Expected Source Code But Executable Program Produces Incorrect Results

Description

The generated source code appears as expected. However, the executable program produces incorrect results.

Action

Do one of the following:

- Lower the compiler optimization level.
 - 1 Select **Custom** for the Model Configuration parameter **Code Generation > Compiler optimization level**.
 - 2 In the **Custom compiler optimization flags** field, specify a lower optimization level.
 - 3 Rebuild the model.
- Disable compiler optimizations.
 - 1 Select **Optimizations off (faster builds)** for the Model Configuration parameter **Code Generation > Compiler optimization level**.
 - 2 Rebuild the model.

For more information, see “Control Compiler Optimizations” on page 31-12 and your compiler documentation.

Compile-Time Issues

Issue	Action
You see a compiler configuration error.	Make sure that MATLAB supports the compiler and version that you want to use. For a list of currently supported and compatible compilers, see www.mathworks.com/support/compilers/current_release/ . If necessary, upgrade or change your compiler (see “Choose and Configure Compiler on Microsoft Windows”

Issue	Action
	on page 25-6 or “Choose and Configure Compiler on UNIX” on page 25-7).
<p>Environment variables for your make utility, compiler, or linker are set up incorrectly. For example, installation of Cygwin tools on a Windows platform affects environment variables used by other compilers.</p>	<p>Review the environment variable settings for your system by using the <code>set</code> command on a Windows platform or <code>setenv</code> on a UNIX platform. Make sure that the settings match what is required for the tools you are using.</p>
<p>Custom code specified as an S-function block or in the Code Generation > Custom Code pane of the Configuration Parameters dialog box includes errors. For example, the code refers to a header file that the compiler cannot find.</p>	<p>Remove the custom code from the model to isolate the source of the problem, debug, and rebuild the model.</p>
<p>The model includes a block, such as a device driver block, which is not intended for use with the currently selected target.</p>	<p>Remove the target-specific block or configure the model for use with the another target.</p>
<p>All of these conditions are true:</p> <ul style="list-style-type: none"> • You represent signal, state, or parameter data by creating a data object such as <code>Simulink.Signal</code>. You use the object in a model reference hierarchy. • You use a custom storage class with the data object. Custom storage classes require an Embedded Coder license. • You set the owner of the object to a model that does not directly access the data. • You use the toolchain <code>lcc-win64</code>. <p>When you generate an executable from the model reference hierarchy, you see a linker error about an undefined reference to the data.</p>	<p>To resolve the issue, choose one of these methods:</p> <ul style="list-style-type: none"> • In the data object, clear the <code>Owner</code> property. Alternatively, set the owner to a model that directly accesses the data. • Use a different toolchain, such as <code>gcc</code>, instead of <code>lcc</code>.

LCC Compiler Does Not Support Ampersands in Source Folder Paths

Description

If you use the LCC compiler and your model folder path contains an ampersand (&), the build process produces an error.

Action

Remove the ampersand from the model folder path. Then, rebuild the model.

More About

- “Choose and Configure Compiler on Microsoft Windows” on page 25-6
- “Choose and Configure Compiler on UNIX” on page 25-7
- “Change Programming Language” on page 25-2
- “Troubleshoot Compiler Issues” on page 25-9
- “Configure Build Process” on page 25-13
- “Initiate the Build Process” on page 25-35
- “Build a Generic Real-Time Program” on page 25-23
- “Rebuild a Model” on page 25-41
- “Reduce Build Time for Referenced Models” on page 25-45
- “Control Regeneration of Top Model Code” on page 25-43
- “Relocate Code to Another Development Environment” on page 25-51
- “Build and Run a Program” on page 25-36
- “Profile Code Performance” on page 25-66
- “Control Compiler Optimizations” on page 31-12
- “Select and Configure Compiler or IDE” on page 25-3
- “How Executable Programs Are Built From Models” on page 25-39
- “Executable Program Generation” on page 25-63

External Websites

- www.mathworks.com/support/compilers/current_release/

Configure Build Process

Choose a Build Process

The code generator supports two processes for building code generated from Simulink models:

- *Toolchain approach* — A newer build process that generates optimized makefiles and supports custom toolchains. The benefits of this approach include:
 - Provide control of your build process with toolchain information objects. These objects can be defined using MATLAB scripts.
 - Supports model referencing, PIL, SIL, and Rapid Accelerator builds.
 - Supported by MATLAB Coder.
 - Provides added flexibility to configure the build process for individual models.
- *Template makefile approach* — An older build process that uses template makefiles

The “System target file” parameter, located on the **Code Generation** pane in the Configuration Parameters dialog box, determines the build process for a model. When the **System target file** is set to:

- `ert.tlc`, `ert_shrlib.tlc`, `grt.tlc`, or any toolchain-compliant system target file, the build process uses the *toolchain approach*. For more information, see “Toolchain Approach” on page 25-13 and see “Support Toolchain Approach with Custom Target” on page 37-81.
- Any non-toolchain-compliant system target files. The build process uses the *template makefile approach*. For more information, see “Template Makefile Approach” on page 25-19.

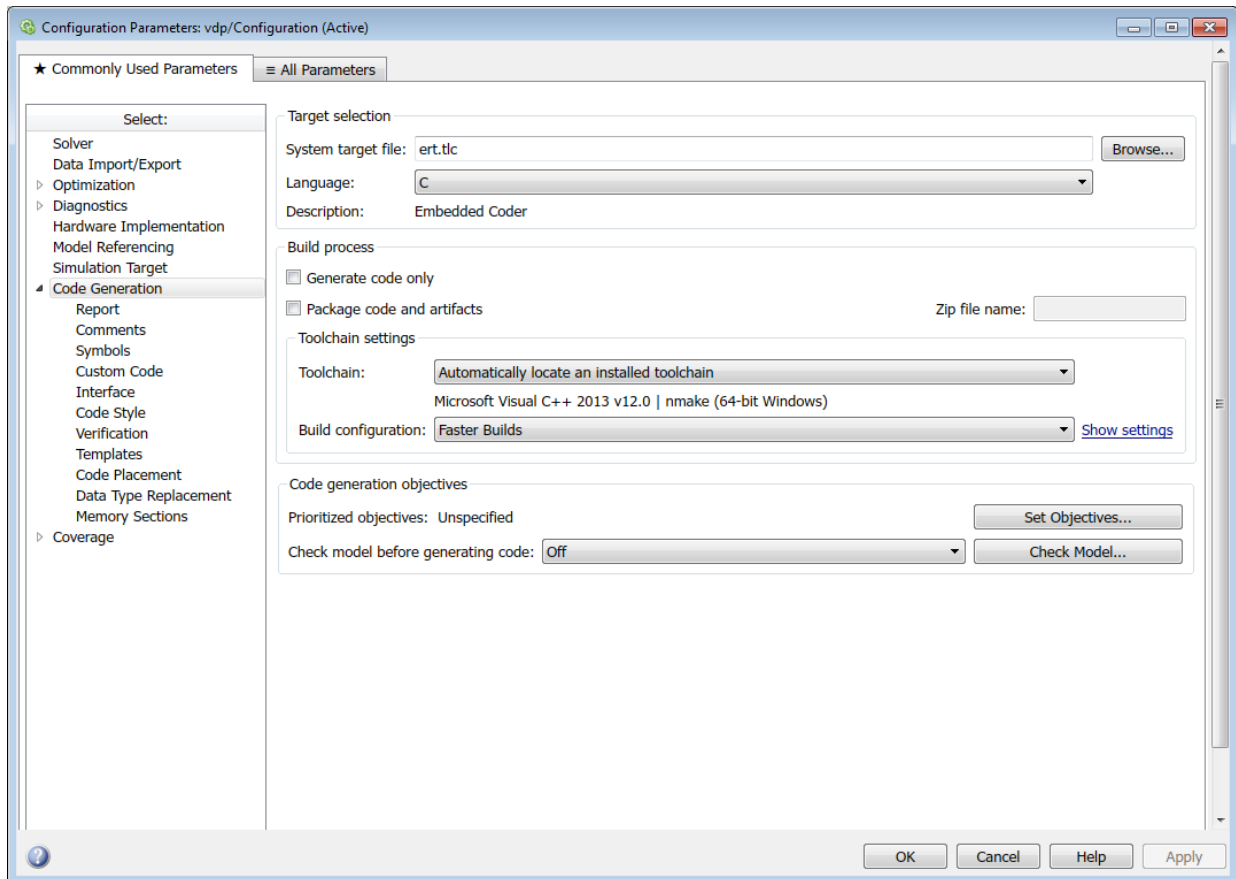
Toolchain Approach

The *toolchain approach* is named for the **Toolchain settings** that appear under **Build process** when you set **System target file** to:

- `grt.tlc` – Generic Real-Time Target
- `ert.tlc` – Embedded Coder (requires the Embedded Coder product)
- `ert_shrlib.tlc` – Embedded Coder (host-based shared library target) (requires the Embedded Coder product)

- Any toolchain-compliant target (If ERT-based, requires the Embedded Coder product)

For more information about toolchain-compliant targets, see “Support Toolchain Approach with Custom Target” on page 37-81.



The **Toolchain settings** include:

- The “Toolchain” parameter specifies the collection of third-party software tools that builds the generated code. A toolchain can include a compiler, linker, archiver, and other prebuild or postbuild tools that download and run the executable on the target hardware.

The default value of **Toolchain** is `Automatically locate an installed toolchain`. The **Toolchain** parameter displays name of the automatically located toolchain just below `Automatically locate an installed toolchain`.

Click the **All Parameters > Toolchain > Validate** button to check that the toolchain is present and validate that the code generator has the information required to use the toolchain. The resulting Validation Report gives a pass/fail for the selected toolchain, and identifies issues to resolve.

- The “Build configuration” parameter lets you choose or customize the optimization settings. By default, **Build Configuration** is set to `Faster Builds`. You can also select `Faster Runs`, `Debug`, and `Specify`. When you select `Specify` and click **Apply**, you can customize the toolchain options for each toolchain. These custom toolchain settings only apply to the current model.

Note: The following system target files, which use the template makefile approach, have the same names but different descriptions from system target files that use the toolchain approach:

- `ert.tlc` – Create Visual C/C++ Solution File for Embedded Coder
- `grt.tlc` – Create Visual C/C++ Solution File for Simulink Coder

To avoid confusion, click **Browse** to select the system target file and look at the description of each file.

Upgrade Model to Use Toolchain Approach

When you open a model created before R2013b that uses the following system target files, the software automatically tries to upgrade the model from using template makefile settings to using the toolchain settings:

- `ert.tlc` – Embedded Coder
- `ert_shrlib.tlc` – Embedded Coder (host-based shared library target)
- `grt.tlc` – Generic Real-Time Target

Note: To upgrade models using a custom system target file to use the toolchain approach, see “Support Toolchain Approach with Custom Target” on page 37-81.

However, some model configuration parameter values prevent the software from automatically upgrading a model to use toolchain settings. The following instructions show you ways to complete the upgrade process.

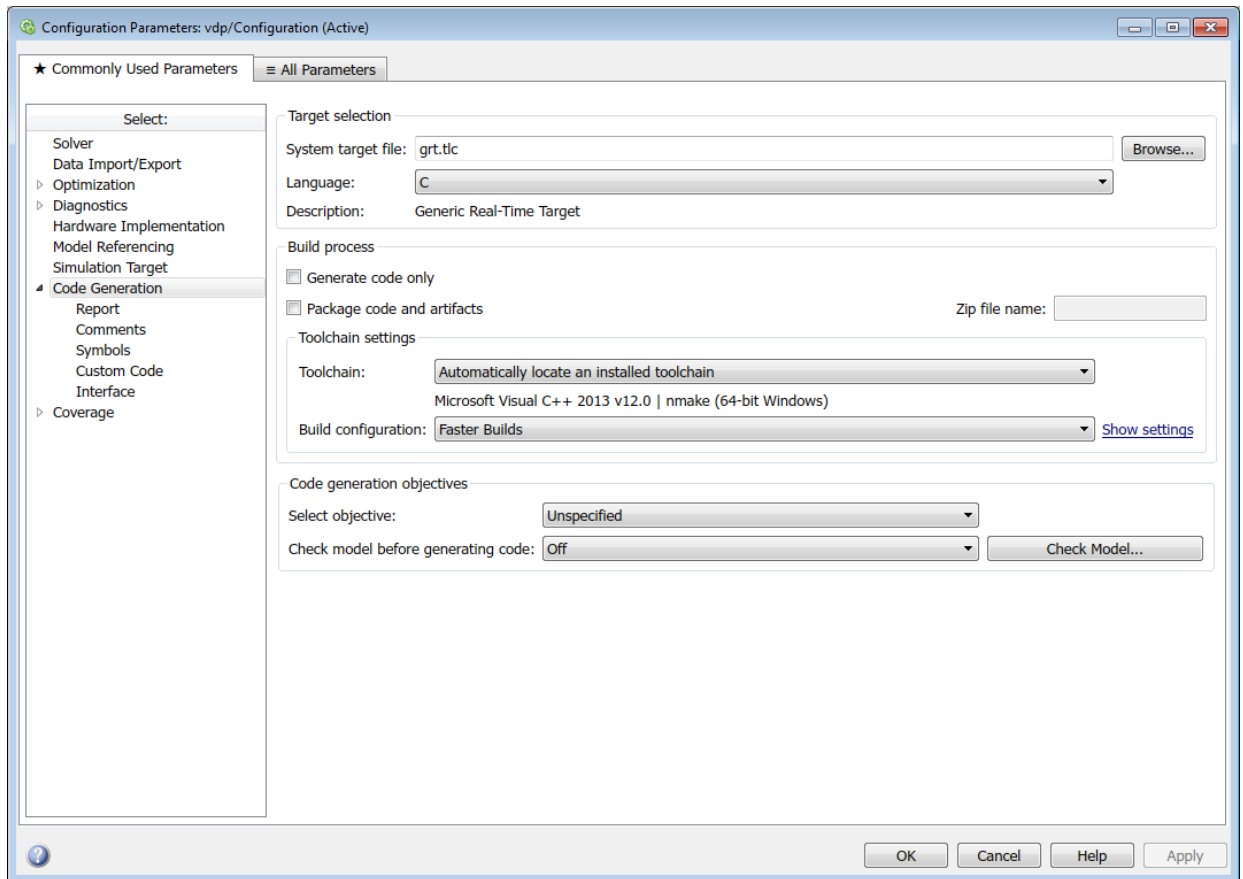
We recommend upgrading your models, but doing so is not required. You can continue generating code from a model that has not been upgraded.

Note: The software does not upgrade models that use the following system target files:

- `ert.tlc` – Create Visual C/C++ Solution File for Embedded Coder
 - `grt.tlc` – Create Visual C/C++ Solution File for Simulink Coder
-

To see if a model was upgraded:

- 1 Open the model configuration parameters by pressing **Ctrl+E**.
- 2 Select the **Code Generation** pane.
- 3 If the **Build process** area contains the **Toolchain** and **Build configuration** parameters, the model has already been upgraded.



If the **Build process** area displays **Makefile configuration** parameters, such as **Generate makefile**, **Make command**, and **Template makefile**, the software has not upgraded the model.

Start by creating a working copy of the model using **File > Save As**. This action preserves the original model and configuration parameters, in case you need to refer back to them.

Try to upgrade the model using Upgrade Advisor:

- 1 In your model, select **Analysis > Model Advisor > Upgrade Advisor**.

- 2 In Upgrade Advisor, select **Check and update model to use toolchain approach to build generated code** and click **Run This Check**.
- 3 Perform the recommended actions and/or click **Update Model**.

When you cannot upgrade the model using Upgrade Advisor, one or more of the following parameters is not set to its default value, shown here:

- **Compiler optimization level** — Optimizations off (faster builds)
- **Generate makefile** — Enabled
- **Template makefile** — Target-specific template makefile
- **Make command** — `make_rtw` without arguments

In some cases, a model cannot be upgraded, however, try the following procedure:

- If **Generate makefile** is disabled, this case might not be upgradable. However, you can try enabling it and try upgrading the model using Upgrade Advisor.
- If **Compiler optimization level** is set to **Optimizations on (faster runs)**:
 - 1 Set **Compiler optimization level** is to **Optimizations off (faster builds)**.
 - 2 Upgrade the model using Upgrade Advisor.
 - 3 Set **Build configuration** to **Faster Runs**.
- If **Compiler optimization level** is set to **Custom**:
 - 1 Copy the **Custom compiler optimization flags** to a text file.
 - 2 Set **Compiler optimization level** to **Optimizations off (faster builds)**.
 - 3 Upgrade the model using Upgrade Advisor.
 - 4 Set **Build configuration** to **Specify**.
 - 5 Edit the compiler options to perform the same optimizations.
- If **Template makefile** uses a customized template makefile, this case might not be upgradable. However, you can try the following:
 - 1 Update **Template makefile** to use the default makefile for the system target file.

Note: To get the default makefile name, change the **System target file**, click **Apply**, change it back, and click **Apply** again.

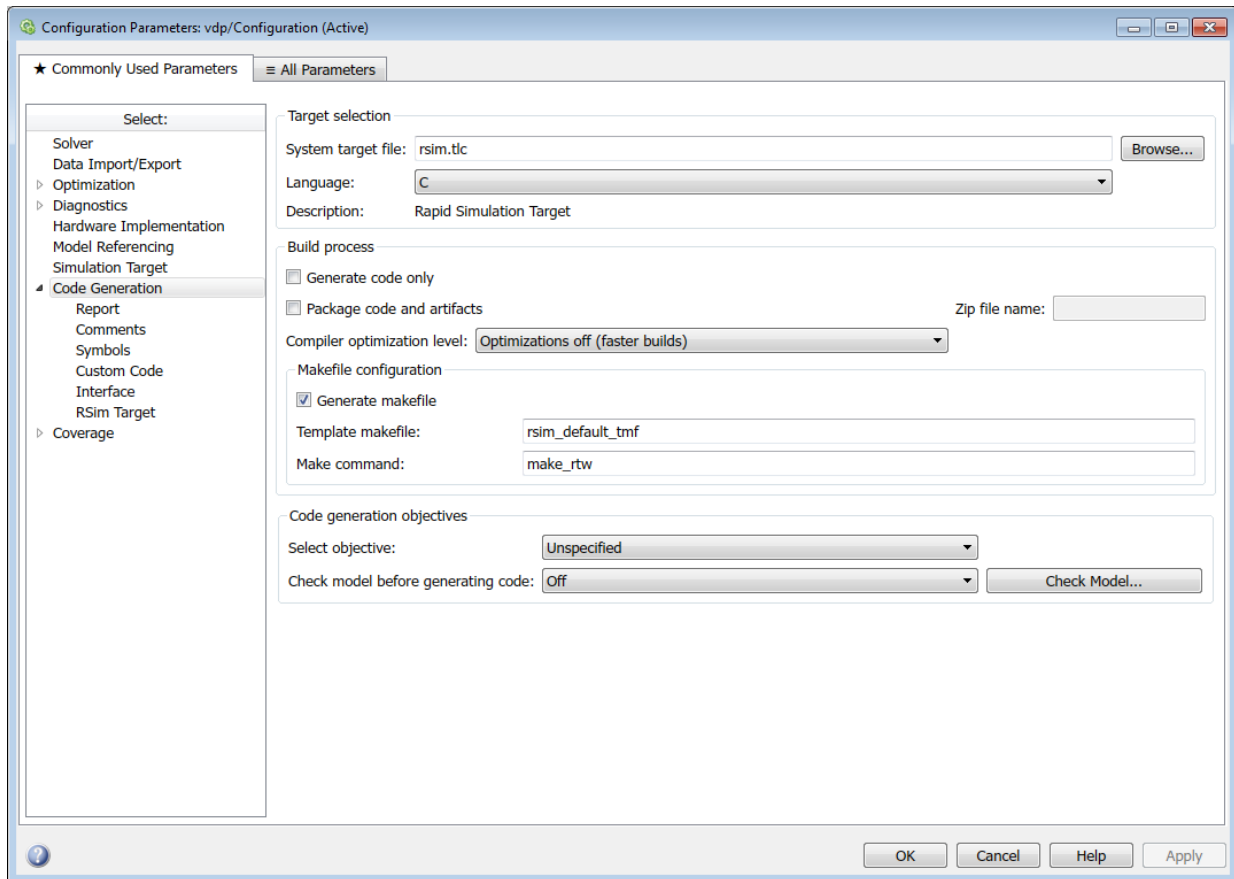
- 2 Upgrade the model using Upgrade Advisor.
- 3 If the template makefile contains build tool options, such as compiler optimization flags, set **Build configuration** to **Specify** and update the options.
- 4 If the template makefile uses custom build tools, create and register a custom toolchain, as described in “Custom Toolchain Registration” . Then, set the **Toolchain** parameter to use the custom toolchain.

Note: After registering the custom toolchain, update **Toolchain** to use the custom toolchain.

- 5 If the template makefile contains custom rules and logic, these cannot be applied to the upgraded model.

Template Makefile Approach

When the **System target file** is set to a `tlc` file that uses the template makefile approach, the software displays **Compiler optimization level**, **Generate makefile**, **Make command**, and **Template makefile** parameters.



Specify Whether To Generate a Makefile

The **Generate makefile** option specifies whether the build process is to generate a makefile for a model. By default, the build process generates a makefile. Suppress generation of a makefile, for example in support of custom build processing that is not based on makefiles, by clearing **Generate makefile**. When you clear this parameter:

- The **Make command** and **Template makefile** options are unavailable.
- You must set up post code generation build processing using a user-defined command, as explained in “Customize Post-Code-Generation Build Processing” on page 35-14.

Specify a Make Command

Each template makefile based target has an associated `make` command, an internal MATLAB command used by code generation software to control the build process. The command displayed in the **Make command** field is invoked when you start a build.

Most targets use the default command, `make_rtw`. Third-party targets might supply another `make` command. See the vendor's documentation.

In addition to the name of the `make` command, you can supply makefile options in the **Make command** field. These options might include compiler-specific options, include paths, and other parameters. When the build process invokes the `make` utility, these options are passed on the `make` command line, which adds them to the overall flags passed to the compiler.

“Template Makefiles and Make Options” on page 17-26 lists the **Make command** options you can use with each supported compiler.

Specify the Template Makefile

The **Template makefile** field has these functions:

- If you have selected a target configuration using the System Target File Browser, this field displays the name of a MATLAB language file that selects a template makefile for your development environment. For example, in “Model Configuration Parameters: Code Generation”, the **Template makefile** field displays `grt_default_tmf`, indicating that the build process invokes `grt_default_tmf.m`.

“Template Makefiles and Make Options” on page 17-26 gives a detailed description of the logic by which the build process selects a template makefile.

- Alternatively, you can explicitly enter the name of a specific template makefile (including the extension) or a MATLAB language file that returns a template makefile in this field. You must do this if you are using a target configuration that does not appear in the System Target File Browser. For example, do this if you have written your own template makefile for a custom target environment.

If you specify your own template makefile, be sure to include the filename extension. If you omit the extension, the build process attempts to find and execute a file with the extension `.m` (that is, a MATLAB language file). The template make file (or a MATLAB language file that returns a template make file) must be on the MATLAB path. To determine whether the file is on the MATLAB path, enter the following command in the MATLAB Command Window:

which *tmf_filename*

Specify TLC Options

You can specify Target Language Compiler (TLC) command line options and arguments for code generation using the model parameter `TLCOptions` in a `set_param` function call. For example,

```
>> set_param(gcs,'TLCOptions','-p0 -aWarnNonSaturatedBlocks=0')
```

Some common uses of TLC options include the following:

- `-aVarName=1` to declare a TLC variable and/or assign a value to it
- `-IC:\Work` to specify an include path
- `-v` to obtain verbose output from TLC processing (for example, when debugging)

TLC options that you specify for code generation are listed in the summary section of the generated HTML code generation report.

Specifying TLC command-line options does not add flags to the make command line.

For additional information, see “Target Language Compiler Overview”.

More About

- “Support Toolchain Approach with Custom Target” on page 37-81
- “Build and Run a Program” on page 25-36
- “Custom Toolchain Registration”
- “Template Makefiles and Make Options” on page 17-26
- “Target Language Compiler Overview”
- “How Executable Programs Are Built From Models” on page 25-39
- “Executable Program Generation” on page 25-63

Build a Generic Real-Time Program

Building a Program

Building a program means generating C or C++ code from an example model and then building an executable program from the generated code. This example uses a Generic Real-Time (GRT) target for code generation. The resulting standalone program runs on your workstation, independent of external timing and events.

Working Folder

This example uses a local copy of the `slexAircraftExample` model, stored in its own folder, `aircraftexample`. Set up your *working folder* as follows:

- 1 In the MATLAB Current Folder browser, navigate to a folder to which you have write access.
- 2 Enter the following MATLAB command to create the working folder:

```
mkdir aircraftexample
```

- 3 Make `aircraftexample` your working folder:

```
cd aircraftexample
```

- 4 Open the `slexAircraftExample` model:

```
slexAircraftExample
```

The model appears in the Simulink Editor model window.

- 5 In the model window, choose **File > Save As**. Navigate to your working folder, `aircraftexample`. Save a copy of the `slexAircraftExample` model as `myAircraftExample`.

Build Folder and Code Generation Folders

While producing code, the code generator creates a *build folder* within your working folder. The build folder name is `model_target_rtw`, derived from the name of the source model and the chosen target. The build folder stores generated source code and other files created during the build process. You examine the build folder contents at the end of this example.

When a model contains Model blocks (references to other models), special *code generation folders* are created in your code generation folder to organize code for the referenced models. Code generation folders exist alongside product build folders and are named `slprj`. “Generate Code for Referenced Models” on page 8-4 describes navigating code generation folder structures in Model Explorer.

Under the `slprj` folder, a subfolder named `_sharedutils` contains generated code that can be shared between models.

Set Simulation Parameters

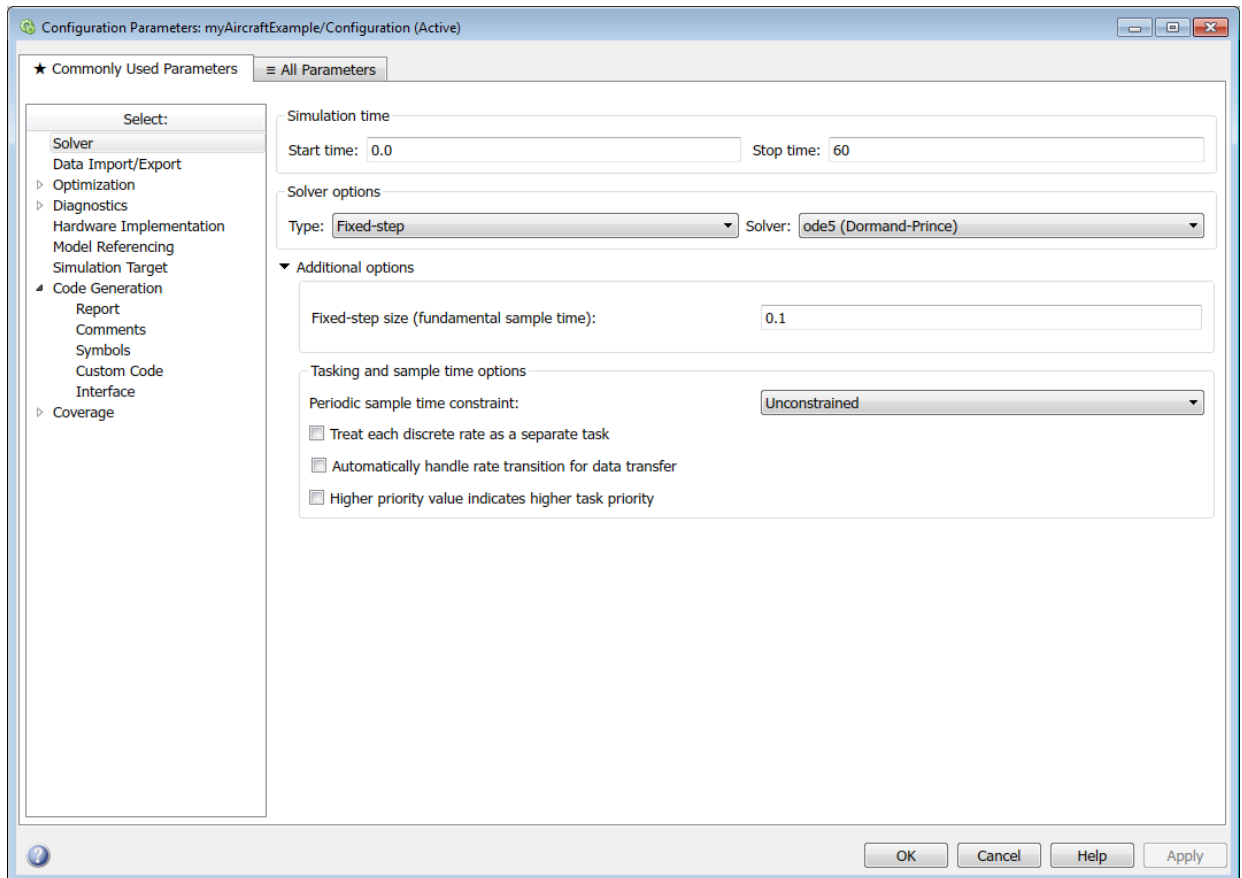
To generate code from your model, you must change some of the simulation parameters. In particular, the generic real-time (GRT) target and most other targets require that the model specify a fixed-step solver.

Note The code generator produces code for models, using variable-step solvers, for rapid simulation (`rsim`) and S-function system target files only.

To set simulation parameters using the Configuration Parameters dialog box:

- 1 Open the `myAircraftExample` model if it is not already open.
- 2 From the model window, open the Configuration Parameters dialog box by selecting **Simulation > Model Configuration Parameters**.
- 3 Select the **Solver** pane. Enter the following parameter values (some might already be set):
 - **Start time:** 0.0
 - **Stop time:** 60
 - **Type:** Fixed-step
 - **Solver:** ode5 (Dormand-Prince)
 - **Fixed step size (fundamental sample time):** 0.1
 - **Treat each discrete rate as a separate task:** Off

The background color of the controls you just changed is tan. The color also appears on fields that were set automatically by your choices in other fields. Use this visual feedback to verify that what you set is what you intended. When you apply your changes, the background color reverts to white.



- 4 Click **Apply** to register your changes.
- 5 Save the model. Simulation parameters persist with the model for use in future sessions.

Select a Target Configuration

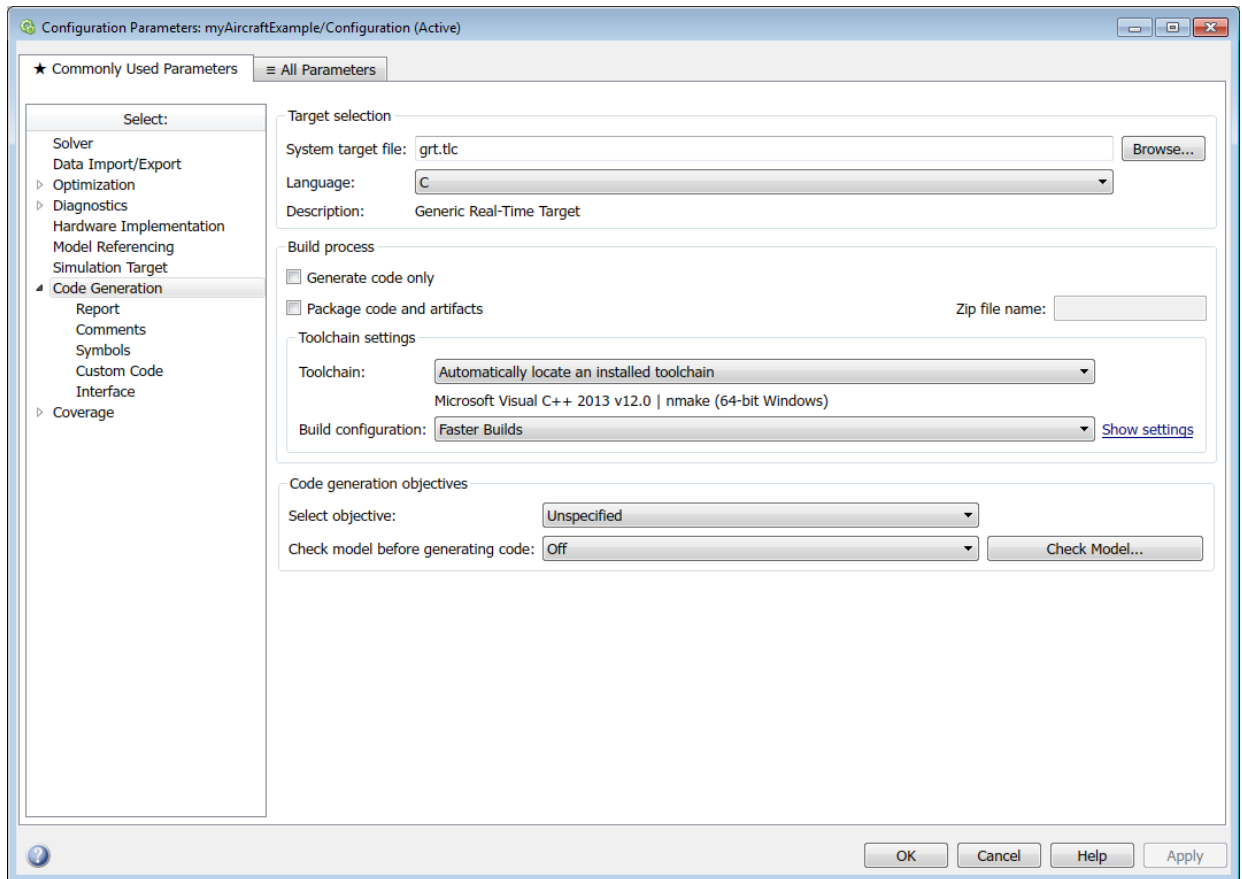
To specify a target configuration for your model, you choose a system target file, a toolchain or template makefile, and a make command.

In these examples and in most applications, you do not need to specify these parameters individually. Here, you use the ready-to-run generic real-time target (GRT) configuration. The GRT target builds a standalone executable program that runs on your workstation.

To select the GRT target using the Configuration Parameters dialog box:

- 1** With the `myAircraftExample` model open, select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box.
- 2** Select the **Code Generation** pane.
- 3** For **System target file**, enter `grt.tlc`, and click **Apply**.

The **Code Generation** pane displays the **System target file** (`grt.tlc`), **Toolchain** (Automatically locate an installed toolchain), and **Build Configuration** (Faster Builds).



Note If you click **Browse**, a System Target File Browser opens and displays the system target files on the MATLAB path. Some system target files require additional licensed products. For example, `ert.tlc` requires the Embedded Coder product.

- 4 Save the model.

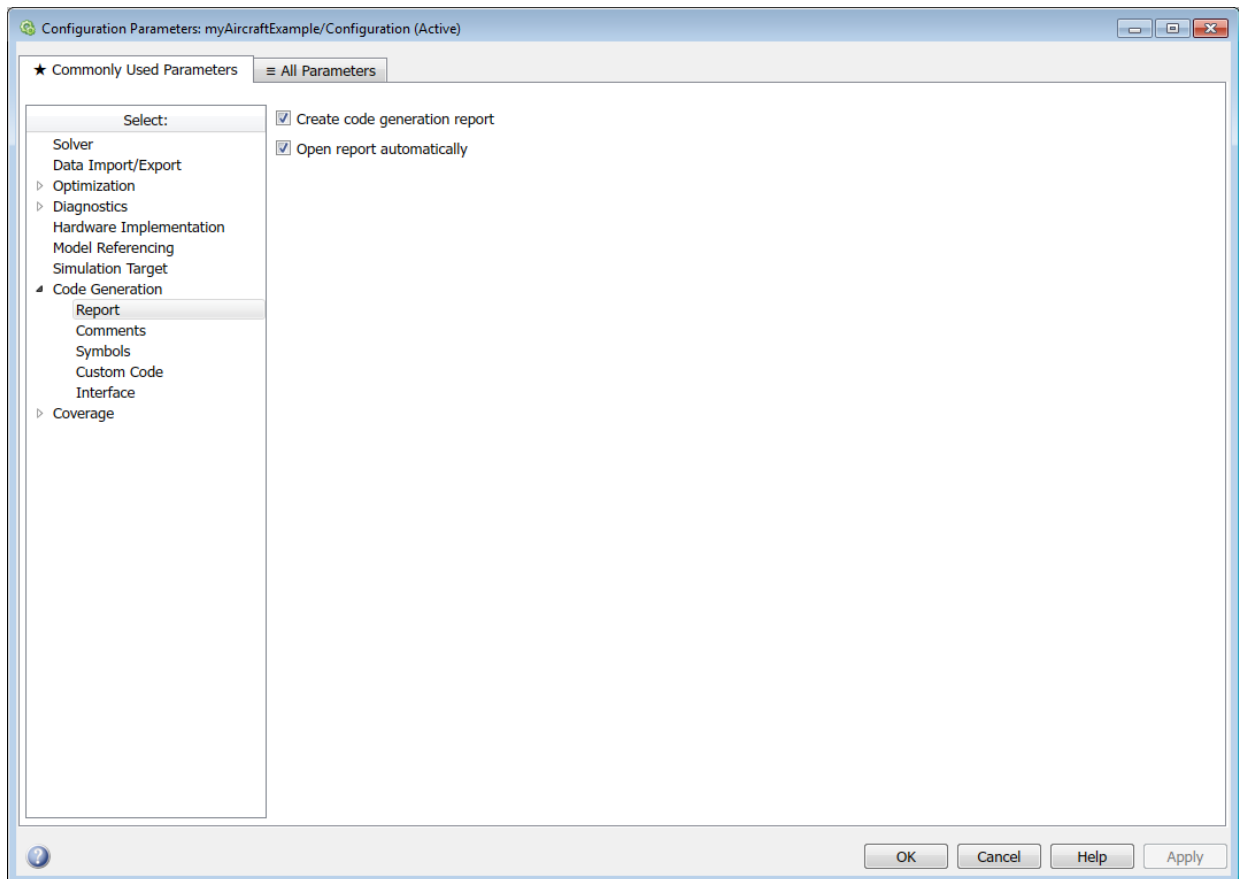
Set Code Generation Parameters

Before you generate code from your model for the first time, inspect the code generation parameters for the model. Some of the steps in this topic do not require you to make

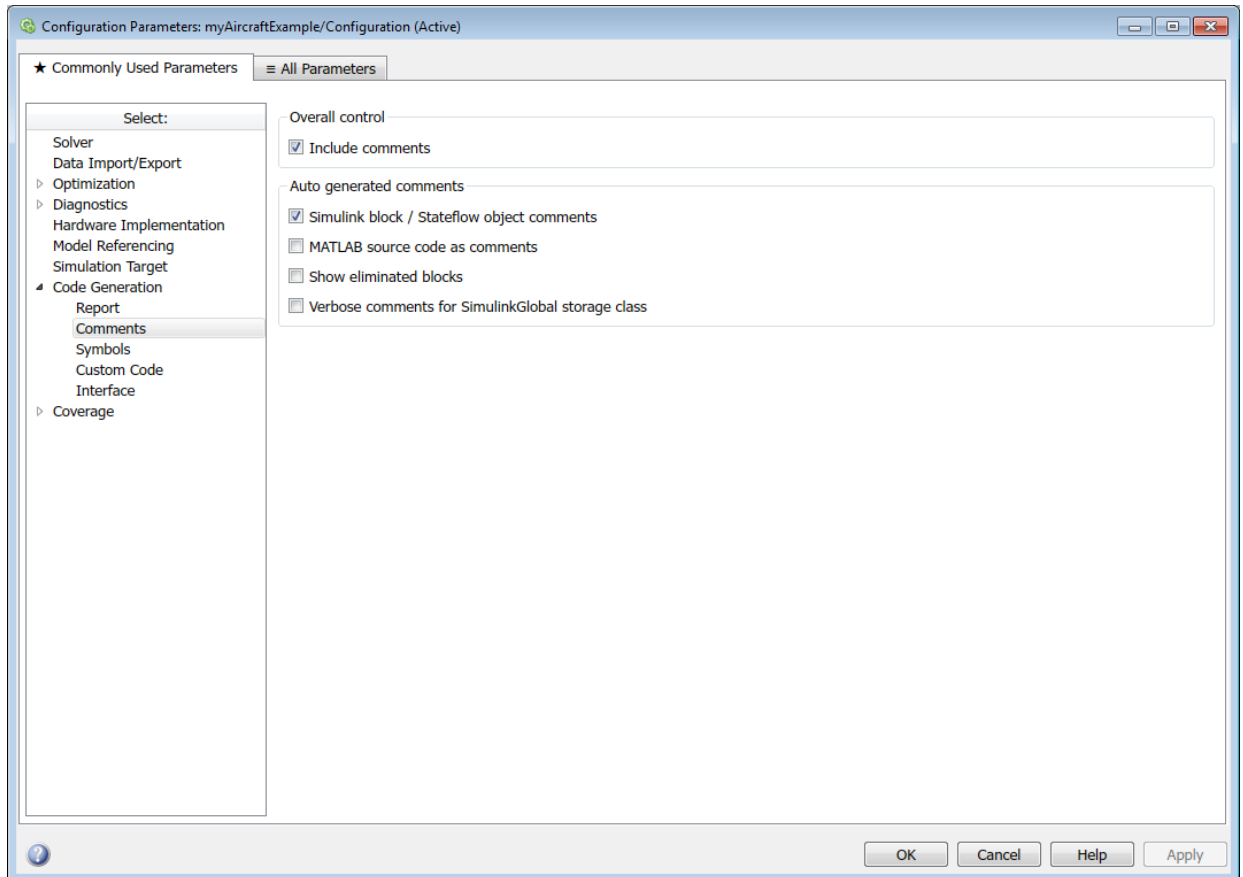
changes. They are included to help you familiarize yourself with the user interface. As you work with the model parameters, you can place the mouse pointer on a parameter to see a tool tip describing its function.

To set code generation parameters using the Configuration Parameters dialog box:

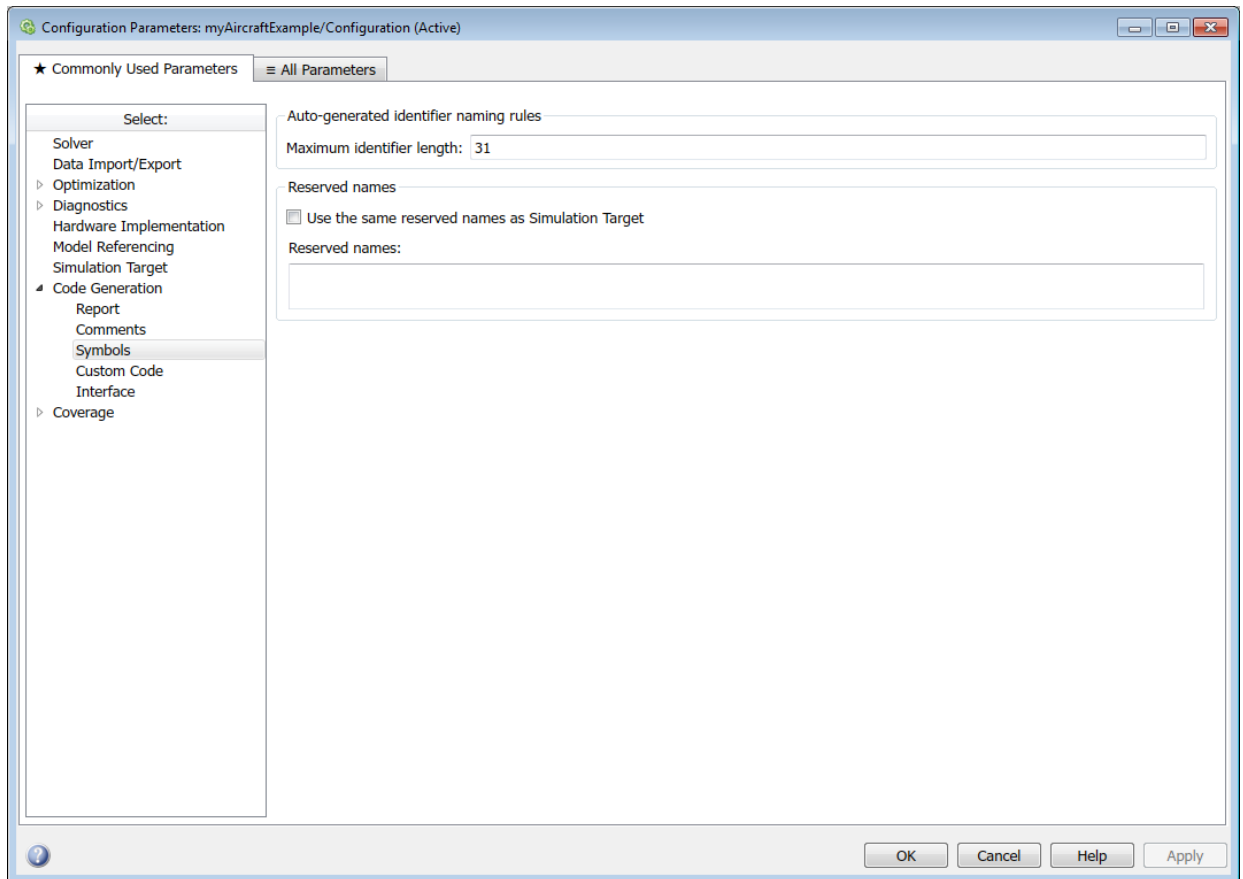
- 1 With the `myAircraftExample` model open, select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box.
- 2 Select **Code Generation > Report > Create code generation report**. This action enables the software to create and display a code generation report for the `myAircraftExample` model.



- 3 Select the **Code Generation > Comments** pane. The options displayed here control the types of comments included in generated code. Leave the options set to their defaults.



- 4 Select the **Code Generation > Symbols** pane. The options on this pane control the look and feel of generated code.



5 Select the **All Parameters** tab. The following parameters in the **Code Generation** category control build verbosity and debugging support. These parameter are common to all target configurations.

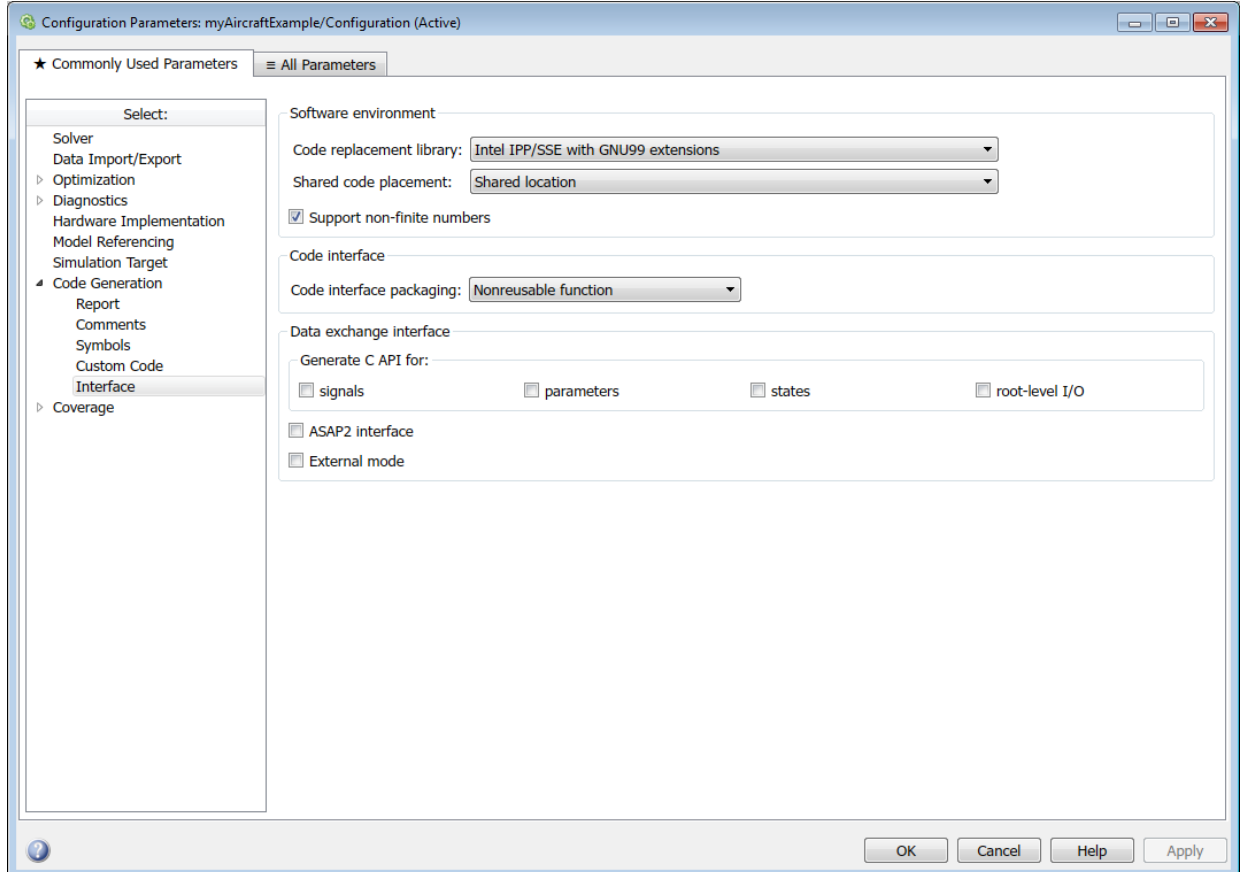
- Verbose build (`RTWVerbose` parameter)
- Retain .rtw file (`RetainRTWFile` parameter)
- Profile TLC (`ProfileTLC` parameter)
- Start TLC debugger when generating code (`TLCDebug` parameter)
- Start TLC coverage when generating code (`TLCCoverage` parameter)

- Enable TLC assertion (TLCAssert parameter)

Leave the options set to their defaults.

6 Select the **Code Generation > Interface** pane.

- For the **Shared code placement** parameter, select **Shared location**. The build process will place generated code for utilities at a shared location within the `slprj` folder in your code generation folder.
- If it is not already cleared, switch to the **All Parameters** tab and clear the **Code Generation > Classic call interface** option.



- 7 Click **Apply** and save the model.

Build and Run a Program

The build process generates C code from the model. It then compiles and links the generated program to create an executable image. To build and run the program:

- 1 With the `myAircraftExample` model open, go to the Configuration Parameters dialog box. In the **Code Generation** pane, click **Build** to start the build process.

A number of messages concerning code generation and compilation appear in the Command Window. The initial message is:

```
### Starting build procedure for model: myAircraftExample
```

The contents of many of the succeeding messages depends on your compiler and operating system. The final messages include:

```
### Created executable myAircraftExample.exe  
### Successful completion of build procedure for model: myAircraftExample  
### Creating HTML report file myAircraftExample_codegen_rpt.html
```

The code generation folder now contains an executable, `myAircraftExample.exe` (Microsoft Windows platforms) or `myAircraftExample` (UNIX platforms). In addition, the build process has created an `slprj` folder and a `myAircraftExample_grt_rtw` folder in your code generation folder.

Note: After generating the code for the `myAircraftExample` model, the build process displays a code generation report. See “Report Generation” for more information about how to create and use a code generation report.

- 2 To see the contents of the working folder after the build, enter the `dir` or `ls` command:

```
>> dir  
  
.  
..  
myAircraftExample.exe  
myAircraftExample.slx  
myAircraftExample.slx.autosave  
myAircraftExample_grt_rtw  
slprj
```

- 3 To run the executable from the Command Window, type `!myAircraftExample`. The `!` character passes the command that follows it to the operating system, which runs the standalone `myAircraftExample` program.

```
>> !myAircraftExample
```

```
** starting the model **
** created myAircraftExample.mat **
```

- 4 To see the files created in the build folder, use the `dir` or `ls` command again. The exact list of files produced varies among MATLAB platforms and versions. Here is a sample list from a Windows platform:

```
>> dir myAircraftExample_grt_rtw

.                rt_main.obj                myAircraftExample_data.c
..               rtmodel.h                myAircraftExample_data.obj
buildInfo.mat    rtw_proj.tmw                myAircraftExample_private.h
codeInfo.mat     myAircraftExample.bat      myAircraftExample_ref.rsp
defines.txt      myAircraftExample.c        myAircraftExample_types.h
html            myAircraftExample.h
modelsources.txt myAircraftExample.mk
rt_logging.obj   myAircraftExample.obj
```

Contents of the Build Folder

The build process creates a build folder and names it *model_target_rtw*, where *model* is the name of the source model and *target* is the target selected for the model. In this example, the build folder is named *myAircraftExample_grt_rtw*.

The build folder includes the following generated files.

File	Description
<code>myAircraftExample.c</code>	Standalone C code that implements the model
<code>myAircraftExample.h</code>	An include header file containing definitions of parameters and state variables
<code>myAircraftExample_private.h</code>	Header file containing common include definitions
<code>myAircraftExample_types.h</code>	Forward declarations of data types used in the code
<code>rtmodel.h</code>	Master header file for including generated code in the static main program (its name does not change, and it simply includes <code>myAircraftExample.h</code>)

The code generation report that you created for the `myAircraftExample` model displays a link for each of these files. You can click the link explore the file contents.

The build folder contains other files used in the build process. They include:

- `myAircraftExample.mk` — Makefile for building executable using the specified Toolchain.
- Object (`.obj`) files
- `myAircraftExample.bat` — Batch control file
- `rtw_proj.tmw` — Marker file
- `buildInfo.mat` — Build information for relocating generated code to another development environment
- `defines.txt` — Preprocessor definitions required for compiling the generated code
- `myAircraftExample_ref.rsp` — Data to include as command-line arguments to `mex` (Windows systems only)

The build folder also contains a subfolder, `html`, which contains the files that make up the code generation report. For more information, see “Reports for Code Generation” on page 20-2.

More About

- “Configure Build Process” on page 25-13
- “Initiate the Build Process” on page 25-35
- “Build and Run a Program” on page 25-36
- “Reports for Code Generation” on page 20-2
- “Select and Configure Compiler or IDE” on page 25-3

Initiate the Build Process

You can initiate code generation and the build process by using the following options:

- Press **Ctrl+B**.
- Select **Code > C/C++ Code > Build Model**.
- Invoke the `rtwbuild` command from the MATLAB command line.
- Invoke the `slbuild` command from the MATLAB command line.

More About

- `rtwbuild`
- “Build a Generic Real-Time Program” on page 25-23
- “Rebuild a Model” on page 25-41
- “Build and Run a Program” on page 25-36

Build and Run a Program

The build process generates C code from a model, and then compiles and links the generated program to create an executable image. To build and run a sample program, use the example model `slexAircraftExample`.

- 1 In the Command Window, enter `slexAircraftExample` to open the model.
- 2 Save a copy of the model to your working folder and name it `myAircraftExample`.
- 3 Open the Configuration Parameters dialog box by selecting **Simulation > Model Configuration Parameters**. Set the following parameters.
 - Select the **Solver** pane. Enter the following parameter values on the **Solver** pane (some may already be set):
 - **Start time:** 0.0
 - **Stop time:** 60
 - **Type:** Fixed-step
 - **Solver:** ode5 (Dormand-Prince)
 - **Fixed step size (fundamental sample time):** 0.1
 - **Treat each discrete rate as a separate task:** Off
 - Select the **Code Generation > Report** pane, and select the **Create code generation report** parameter. This causes the build process to display a code generation report after generating the code for the `myAircraftExample` model.
 - Select the **Code Generation > Interface** pane. For the **Shared code placement** parameter, select `Shared location`. This causes generated code for utilities to be placed at a shared location within the `slprj` folder in the code generation folder.
 - Select the **All Parameters** tab.
 - If it is not already cleared, clear the **Classic call interface** option.
 - If it is not already set, set the **Single output/update function** option.

Click **Apply**.

- 4 In the **Code Generation** pane, click the **Build** button to start the build process.

A number of messages concerning code generation and compilation appear in the MATLAB Command Window. The initial message is

```
### Starting build procedure for model: myAircraftExample
```

The contents of many of the succeeding messages depends on your compiler and operating system. The final messages include

```
### Created executable slxAircraftExample.exe
### Successful completion of build procedure for model: myAircraftExample
### Creating HTML report file myAircraftExample_codegen_rpt.html
```

The code generation folder now contains an executable, `myAircraftExample.exe` (Microsoft Windows platforms) or `myAircraftExample` (UNIX platforms). In addition, the build process has created an `slprj` folder and a `myAircraftExample_grt_rtw` folder in the code generation folder .

Note: The build process displays a code generation report after generating the code for the `myAircraftExample` model. See “Report Generation” for more information about how to create and use a code generation report.

- 5 To observe the contents of the working folder after the build, type the `dir` or `ls` command from the Command Window.

```
>> dir
```

```
.                myAircraftExample.exe      myAircraftExample_grt_rtw
..               myAircraftExample.slx    slprj
```

- 6 To run the executable from the Command Window, type `!slxAircraftExample`. The `!` character passes the command that follows it to the operating system, which runs the standalone `slxAircraftExample` program.

```
>> !myAircraftExample
```

```
** starting the model **
** created myAircraftExample.mat **
```

- 7 To see the files created in the build folder, use the `dir` or `ls` command again. The exact list of files produced varies among MATLAB platforms and versions. Here is a sample list from a Windows platform.

```
>> dir myAircraftExample_grt_rtw
```

```
.                rt_main.obj                myAircraftExample_data.c
..               rtmodel.h                myAircraftExample_data.obj
buildInfo.mat    rtw_proj.tmw                myAircraftExample_private.h
codeInfo.mat     myAircraftExample.bat      myAircraftExample_ref.rsp
defines.txt      myAircraftExample.c        myAircraftExample_types.h
html             myAircraftExample.h
modelsources.txt myAircraftExample.mk
```

rt_logging.obj myAircraftExample.obj

Tip For UNIX platforms, run the executable in the Command Window with the syntax `!./executable_name`. If preferred, run the executable from an OS shell with the syntax `./executable_name`. For more information, see “Run External Commands, Scripts, and Programs”.

More About

- “Run External Commands, Scripts, and Programs”
- “Initiate the Build Process” on page 25-35
- “Rebuild a Model” on page 25-41
- “Report Generation”

How Executable Programs Are Built From Models

The code generator produces C code only or generates the C code and produces an executable image, depending on the level of processing you choose. By default, the software generates the C code and produces an executable image. If you select the **Generate code only** check box, the software generates code only.

Build Process Steps

When you press **Ctrl+B**, the code generator performs the following build process. If the software detects code generation constraints for your model, it issues warning or error messages.

- 1 “Model Compilation” on page 19-71
- 2 “Code Generation” on page 19-71
- 3 “Customized Makefile Generation” on page 25-39
- 4 “Executable Program Generation” on page 25-63

For more information, see “Generate a Code Generation Report” on page 20-5. You can also view an HTML report in Model Explorer.

Customized Makefile Generation

After producing code, the code generator produces a customized makefile, *model.mk*. The generated makefile instructs the **make** system utility to compile and link source code generated from the model, as well as any required harness program, libraries, or user-provided modules. The code generator produces the file *model.mk* whether you are using the toolchain approach or template makefile approach for build process control. For more information about these approaches, see “Configure Build Process” on page 25-13.

If you are using the template makefile approach, the code generator creates *model.mk* from a system template file, *system.tmf* (where *system* stands for the selected target name). The system template makefile is designed for your target environment. You have the option of modifying the template makefile to specify compilers, compiler options, and additional information used during the creation of the executable. For more information, see “Template Makefile Approach” on page 25-19.

If you are using the toolchain approach, the code generator creates *model.mk* based on the model **Toolchain settings**. You can modify generation of the makefile through the `rtwmakecfg.m` API. For more information, see “Toolchain Approach” on page 25-13.

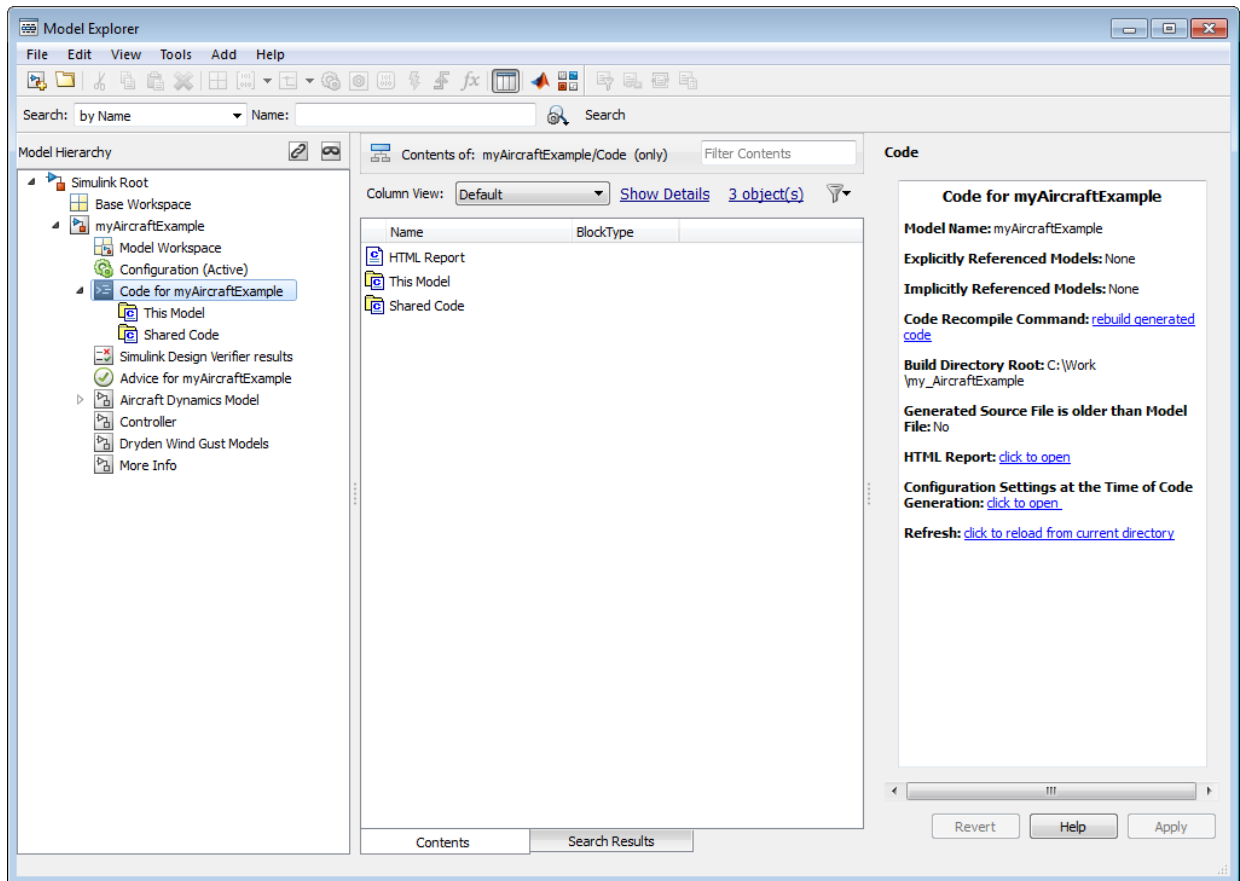
More About

- “Initiate the Build Process” on page 25-35
- “Control Regeneration of Top Model Code” on page 25-43
- “Reduce Build Time for Referenced Models” on page 25-45
- “Relocate Code to Another Development Environment” on page 25-51
- “Build and Run a Program” on page 25-36

Rebuild a Model

If you update generated source code or makefiles manually to add customizations, you can rebuild the files with the `rtwrebuild` command. This command recompiles the modified files by invoking the generated makefile. Alternatively, you can use this command from the Model Explorer:

- 1 Open the top model. Select **View > Model Explorer** to open the Model Explorer window.
- 2 In the **Model Hierarchy** pane, expand the node for the model.
- 3 Click the **Code for *model*** node.
- 4 In the **Code** pane, next to **Code Recompile Command**, click **rebuild generated code**.



More About

- `rtwrebuild`
- “Initiate the Build Process” on page 25-35
- “Build and Run a Program” on page 25-36

Control Regeneration of Top Model Code

When you rebuild a model, by default, the build process performs checks to determine whether changes to the model or relevant settings require regeneration of the top model code. Top model code is regenerated if any of the following conditions is true:

- The structural checksum of the model has changed.
- The top-model-only checksum has changed. The top-model-only checksum provides information about top model parameters, such as application lifespan, maximum stack size, make command, verbose and `.rtw` file debug settings, and `TLCOptions`.
- Any of the following TLC debugging model options, located on the **All Parameters** tab of the Configuration Parameters dialog box, is on:
 - **Start TLC debugger when generating code** (`TLCDebug`)
 - **Start TLC coverage when generating code** (`TLCCoverage`)
 - **Enable TLC assertion** (`TLCAssert`)
 - **Profile TLC** (`ProfileTLC`)

If the checks determine that top model code generation is required, the build process fully regenerates and compiles the model code.

If the checks indicate that the top model generated code is current with respect to the model, and no model settings require full regeneration, the build process omits regeneration of the top model code. This can significantly reduce model build times.

Note: With an Embedded Coder license, if you modify a code generation template (CGT) file and then rebuild your model, the code generation process does not force a top model build. In this case, see “Force Regeneration of Top Model Code” on page 25-44.

Regardless of whether the top model code is regenerated, the build process subsequently calls the build process hooks, including `STF_make_rtw_hook` functions and the post code generation command, and reruns the makefile so that external dependencies are recompiled and relinked.

Note: Target authors can perform actions related to code regeneration, including forcing or reacting to code regeneration, in the `STF_make_rtw_hook` functions that are called

by the build process. For more information, see “Control Code Regeneration Using `STF_make_rtw_hook.m`” on page 35-34.

Force Regeneration of Top Model Code

If you want to control or override the default top model build behavior, use one of the following command-line options:

- To ignore the checksum and force regeneration of the top model code:
 - `rtwbuild(model, 'ForceTopModelBuild', true)`
 - `slbuild(model, 'StandaloneRTWTarget', 'ForceTopModelBuild', true)`
- To clean the model build area enough to trigger regeneration of the top model code at the next build (slbuild only):
 - `slbuild(model, 'CleanTopModel')`

Note: You can also force regeneration of the top model code by deleting the `slprj` folder or the generated model code folder from the code generation folder .

More About

- `rtwrebuild`
- “Control Code Regeneration Using `STF_make_rtw_hook.m`” on page 35-34
- “Configure Build Process” on page 25-13
- “Rebuild a Model” on page 25-41
- “Reduce Build Time for Referenced Models” on page 25-45

Reduce Build Time for Referenced Models

In a parallel computing environment, you can increase the speed of code generation and compilation for models containing large model reference hierarchies by building referenced models in parallel whenever conditions allow. For example, if you have Parallel Computing Toolbox™ software, code generation and compilation for each referenced model can be distributed across the cores of a multicore host computer. If you additionally have MATLAB Distributed Computing Server™ software, code generation and compilation for each referenced model can be distributed across remote workers in your MATLAB Distributed Computing Server configuration.

Parallel Building For Large Model Reference Hierarchies

The performance gain realized by using parallel builds for referenced models depends on several factors, including how many models can be built in parallel for a given model referencing hierarchy, the size of the referenced models, and parallel computing resources such as number of local and/or remote workers available and the hardware attributes of the local and remote machines (amount of RAM, number of cores, and so on).

For configuration requirements that might apply to your parallel computing environment, see “Parallel Building Configuration Requirements” on page 25-46.

For a description of the general workflow for building referenced models in parallel whenever conditions allow, see “Build Models In a Parallel Computing Environment” on page 25-46.

For information on how to configure a custom embedded target to support parallel builds, see “Support Model Referencing” on page 37-83.

Note: In a MATLAB Distributed Computing Server parallel computing configuration, parallel building is designed to work interactively with the software. You can initiate builds from the Simulink user interface or from the MATLAB Command Window using commands such as `slbuild`. You cannot initiate builds using `batch` or other batch mode workflows.

Parallel Building Configuration Requirements

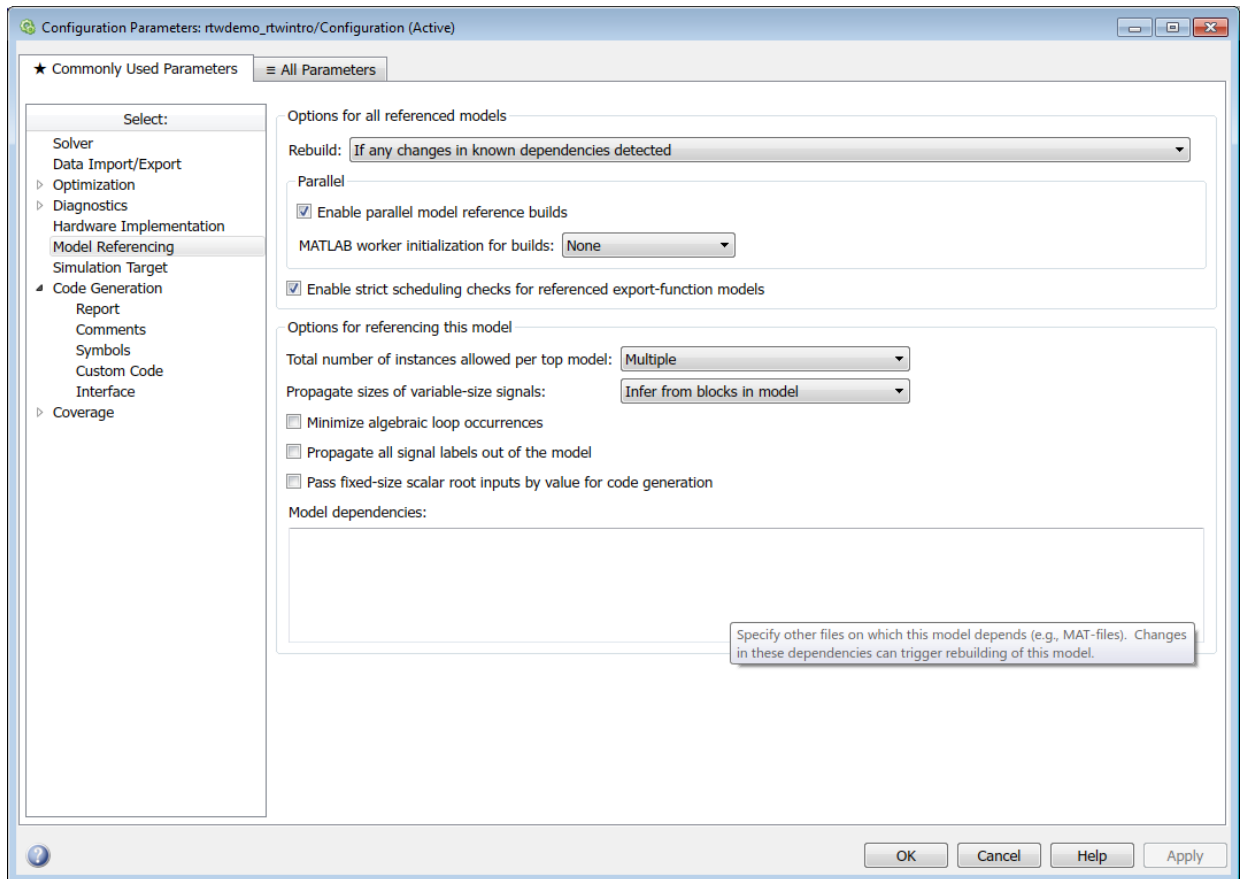
The following requirements apply to configuring your parallel computing environment for building model reference hierarchies in parallel whenever conditions allow:

- For local pools, the host machine must have enough RAM to support the number of local workers (MATLAB sessions) that you plan to use. For example, using `parpool(4)` to create a parallel pool with four workers results in five MATLAB sessions on your machine, each using the amount of memory required for MATLAB and the code generator at startup. For memory requirements, see System Requirements for MATLAB & Simulink.
- Remote MATLAB Distributed Computing Server workers participating in a parallel build and the client machine must use a common platform and compiler.
- A consistent MATLAB environment must be set up in each MATLAB worker session as in the MATLAB client session — for example, shared base workspace variables, MATLAB path settings, and so forth. One approach is to use the `PreLoadFcn` callback of the top model. If you configure your model to load the top model with each MATLAB worker session, its preload function can be used for any MATLAB worker session setup.

Build Models In a Parallel Computing Environment

To take advantage of parallel building for a model reference hierarchy:

- 1 Set up a pool of local and/or remote MATLAB workers in your parallel computing environment.
 - a Make sure that Parallel Computing Toolbox software is licensed and installed.
 - b To use remote workers, make sure that MATLAB Distributed Computing Server software is licensed and installed.
 - c Issue MATLAB commands to set up the parallel pool, for example, `parpool(4)`.
- 2 From the top model of the model reference hierarchy, open the Configuration Parameters dialog box. Go to the **Model Referencing** pane and select the **Enable parallel model reference builds** option. This selection enables the parameter **MATLAB worker initialization for builds**.



For **MATLAB worker initialization for builds**, select one of the following values:

- **None** if the software should not perform special worker initialization. Specify this value if the child models in the model reference hierarchy do not rely on anything in the base workspace beyond what they explicitly set up (for example, with a model load function).
- **Copy base workspace** if the software should attempt to copy the base workspace to each worker. Specify this value if you use a setup script to prepare the base workspace for multiple models to use.

- `Load top model` if the software should load the top model on each worker. Specify this value if the top model in the model reference hierarchy handles the base workspace setup (for example, with a model load function).

Note: You only need to set **Enable parallel model reference builds** for the top model of the model reference hierarchy to which it applies.

- 3 Optionally, turn on verbose messages for simulation builds, code generation builds, or both. If you select verbose builds, the build messages report the progress of each parallel build with the name of the model.
 - To turn on verbose messages for simulation target builds, go to the **All Parameters** tab of the Configuration Parameters dialog box and select **Verbose accelerator builds**.
 - To turn on verbose messages for code generation target builds, go to the **Debug** pane of the Configuration Parameters dialog box and select **Verbose build**

Both options control the verbosity of build messages both in the MATLAB Command Window and in parallel build log files.

- 4 Optionally, inspect the model reference hierarchy to determine, based on model dependencies, which models will be built in parallel. For example, you can use the Model Dependency Viewer from the Simulink **Analysis > Model Dependencies** menu.
- 5 Build your model. Messages in the MATLAB Command Window record when each parallel or serial build starts and finishes. The order in which referenced models build is nondeterministic. They might build in a different order each time the model is built.

If you need more information about a parallel build, for example, if a build fails, see “Locate Parallel Build Logs” on page 25-48.

Locate Parallel Build Logs

When you build a model for which referenced models are built in parallel, if verbose builds are turned on, messages in the MATLAB Command Window record when each parallel or serial build starts and finishes. For example,

```
### Initializing parallel workers for parallel model reference build.  
### Parallel worker initialization complete.  
### Starting parallel model reference SIM build for 'bot_model1001'
```

```

### Starting parallel model reference SIM build for 'bot_model002'
### Starting parallel model reference SIM build for 'bot_model003'
### Starting parallel model reference SIM build for 'bot_model004'
### Finished parallel model reference SIM build for 'bot_model001'
### Finished parallel model reference SIM build for 'bot_model002'
### Finished parallel model reference SIM build for 'bot_model003'
### Finished parallel model reference SIM build for 'bot_model004'
### Starting parallel model reference RTW build for 'bot_model001'
### Starting parallel model reference RTW build for 'bot_model002'
### Starting parallel model reference RTW build for 'bot_model003'
### Starting parallel model reference RTW build for 'bot_model004'
### Finished parallel model reference RTW build for 'bot_model001'
### Finished parallel model reference RTW build for 'bot_model002'
### Finished parallel model reference RTW build for 'bot_model003'
### Finished parallel model reference RTW build for 'bot_model004'

```

To obtain more detailed information about a parallel build, you can examine the parallel build log. For each referenced model built in parallel, the build process generates a file named *model_buildlog.txt*, where *model* is the name of the referenced model. This file contains the full build log for that model.

If a parallel build completes, you can find the build log file in the build subfolder corresponding to the referenced model. For example, for a build of referenced model *bot_model004*, look for the build log file *bot_model004_buildlog.txt* in a referenced model subfolder such as *build_folder/slprj/grt/bot_model004*, *build_folder/slprj/ert/bot_model004*, or *build_folder/slprj/sim/bot_model004*. The build log (diagnostic viewer) provides a relative path to the location of each build log file.

If a parallel builds fails, you might see output similar to the following:

```

### Initializing parallel workers for parallel model reference build.
### Parallel worker initialization complete.
...
### Starting parallel model reference RTW build for 'bot_model002'
### Starting parallel model reference RTW build for 'bot_model003'
### Finished parallel model reference RTW build for 'bot_model002'
### Finished parallel model reference RTW build for 'bot_model003'
### Starting parallel model reference RTW build for 'bot_model001'
### Starting parallel model reference RTW build for 'bot_model004'
### Finished parallel model reference RTW build for 'bot_model004'
### The following error occurred during the parallel model reference RTW build for
'bot_model001':

Error(s) encountered while building model "bot_model001"

### Cleaning up parallel workers.

```

If a parallel build fails, you can find the build log file in a referenced model subfolder under the build subfolder */par_md1_ref/model*. For example, for a failed parallel build of model *bot_model001*, look for the build log file *bot_model001_buildlog.txt*

in a subfolder such as *build_folder/par_md1_ref/bot_model001/slprj/grt/bot_model001*, *build_folder/par_md1_ref/bot_model001/slprj/ert/bot_model001*, or *build_folder/par_md1_ref/bot_model001/slprj/sim/bot_model001*.

More About

- “Initiate the Build Process” on page 25-35
- “Control Regeneration of Top Model Code” on page 25-43
- “Reduce Update Time for Referenced Models”
- “Build and Run a Program” on page 25-36
- “Profile Code Performance” on page 25-66
- “Support Model Referencing” on page 37-83

Relocate Code to Another Development Environment

If you need to relocate the static and generated code files for a model to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB and Simulink products, use the pack-and-go utility.

Code Relocation

The pack-n-go utility uses the tools for customizing the build process after code generation and a `packNGo` function to find and package files for building an executable image. The files are packaged in a compressed file that you can relocate and unpack using a standard `zip` utility.

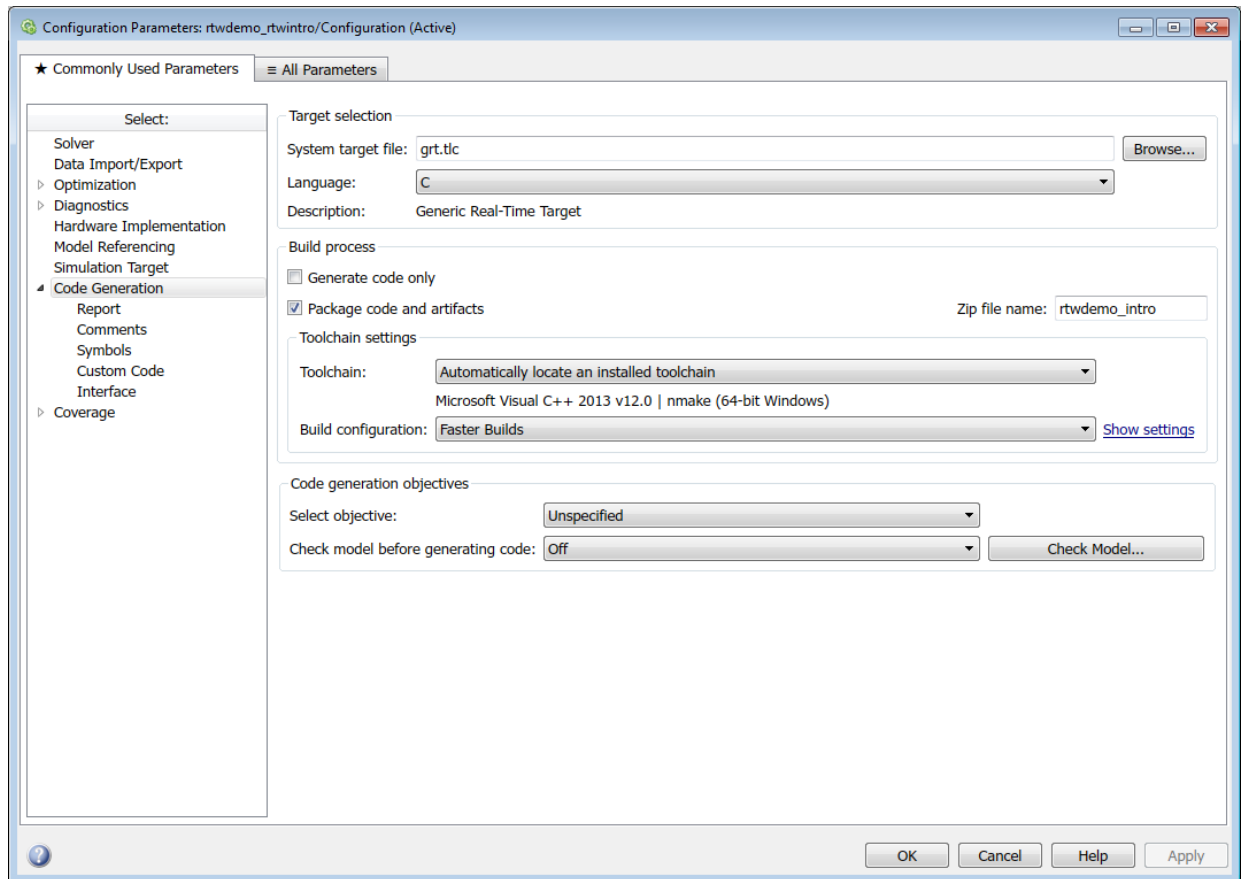
To package model code files, you can do either of the following:

- Use the model option **Package code and artifacts** on the **Code Generation** pane of the Configuration Parameters dialog box. See “Package Code Using the Graphical User Interface” on page 25-51.
- Use MATLAB commands to configure a `PostCodeGenCommand` parameter with a call to the `packNGo` function. See “Package Code Using the Command-Line Interface” on page 25-52. The command-line interface provides more control over the details of code packaging.

Package Code Using the Graphical User Interface

To package and relocate code for your model using the graphical user interface:

- 1 Open the Configuration Parameters dialog box and select the **Code Generation** pane.
- 2 Select the option **Package code and artifacts**. This option configures the build process to run the `packNGo` function after code generation to package generated code and artifacts for relocation.
- 3 In the **Zip file name** field, enter the name of the `zip` file in which to package generated code and artifacts for relocation. You can specify the file name with or without the `.zip` extension. If you specify no extension or an extension other than `.zip`, the `zip` utility adds the `.zip` extension. If you do not specify a value, the build process uses the name `model.zip`, where `model` is the name of the top model for which code is being generated.



- 4 Apply changes and generate code for your model. Inspect the resulting zip file to verify that it is ready for relocation. Depending on the zip tool that you use, you might be able to open and inspect the file without unpacking it.
- 5 Relocate the zip file to the destination development environment and unpack the file.

Package Code Using the Command-Line Interface

To package and relocate code for your model using the command-line interface:

- 1 Select a structure for the zip file.

- 2 Select a name for the zip file.
- 3 Package the model code files in the zip file.
- 4 Inspect the generated zip file.
- 5 Relocate and unpack the zip file.

Select a Structure for the Zip File

Before you generate and package the files for a model build, decide whether you want the files to be packaged in a flat or hierarchical folder structure. By default, the `packNGo` function packages the files in a single, flat folder structure.

If...	Then Use a...
You are relocating files to an IDE that does not use the generated makefile, or the code is not dependent on the relative location of required static files	Single, flat folder structure
The target development environment must maintain the folder structure of the source environment because it uses the generated makefile, or the code is dependent on the relative location of files	Hierarchical structure

If you use a hierarchical structure, the `packNGo` function creates two levels of zip files, a primary zip file, which in turn contains the following secondary zip files:

- `mlrFiles.zip` — files in your *matlabroot* folder tree
- `sDirFiles.zip` — files in and under your build folder where you initiated the model's code generation
- `otherFiles.zip` — required files not in the *matlabroot* or *start* folder trees

Paths for the secondary zip files are relative to the root folder of the primary zip file, maintaining the source development folder structure.

Select a Name for the Zip File

By default, the `packNGo` function names the primary zip file *model*.. You have the option of specifying a different name. If you specify a file name and omit the file type extension, the function appends `.zip` to the name that you specify.

Package Model Code in a Zip File

You package model code files by using the `PostCodeGenCommand` configuration parameter, `packNGo` function, and the model's build information object. You can set up the packaging operation to use:

- A system generated build information object.

In this case, before generating the model code, use `set_param` to set the configuration parameter `PostCodeGenCommand` to an explicit call to the `packNGo` function. For example:

```
set_param(bdroot, 'PostCodeGenCommand', 'packNGo(buildInfo);');
```

This command instructs the build process to evaluate the call to `packNGo`, using the system generated build information object for the currently selected model, after generating and writing the model code to disk and before generating a makefile.

- A build information object that you construct programmatically.

In this case, you might use other build information functions to selectively include paths and files in the build information object that you then specify with the `packNGo` function. For example:

```
.
.
.
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo, {'test1.c' 'test2.c' 'driver.c'});
.
.
.
packNGo(myModelBuildInfo);
```

The following examples show how you can change the default behavior of `packNGo`.

To...	Specify...
Change the structure of the file packaging to hierarchical	<code>packNGo(buildInfo, {'packType' 'hierarchical'});</code>
Rename the primary zip file	<code>packNGo(buildInfo, {'fileName' 'zippedsrcs'});</code>
Change the structure of the file packaging to hierarchical and rename the primary zip file	<code>packNGo(buildInfo, {'packType' 'hierarchical' ...</code>

To...	Specify...
	<code>'fileName' 'zippedsrcs'});</code>
Include header files found on the include path in the zip file	<code>packNGO(buildInfo, {'minimalHeaders' false});</code>
Generate warnings for parse errors and missing files	<code>packNGO(buildInfo, {'ignoreParseError' true... 'ignoreFileMissing' true});</code>

Note: The `packNGO` function can potentially modify the build information passed in the first `packNGO` argument. As part of packaging model code, `packNGO` might find additional files from source and include paths recorded in the model's build information and add them to the build information.

Inspect a Generated Zip File

Inspect the generated zip file to verify that it is ready for relocation. Depending on the zip tool that you use, you might be able to open and inspect the file without unpacking it. If you need to unpack the file and you packaged the model code files as a hierarchical structure, you will need to unpack the primary and secondary zip files. When you unpack the secondary zip files, relative paths of the files are preserved.

Relocate and Unpack a Zip File

Relocate the generated zip file to the destination development environment and unpack the file.

Code Packaging Example

This example shows how to package code files generated for the example model `rtwdemo_rtwinintro` using the command-line interface:

- 1 Set your working folder to a writable folder.
- 2 Open the model `rtwdemo_rtwinintro` and save a copy to your working folder.
- 3 Enter the following MATLAB command:

```
set_param('rtwdemo_rtwinintro', 'PostCodeGenCommand', ...
'packNGO(buildInfo, {'packType' 'hierarchical'}));
```

You must double the single-quotes due to the nesting of character arrays 'packType' and 'hierarchical' within the character array that specifies the call to packNGo.

- 4 Generate code for the model.
- 5 Inspect the generated zip file, `rtwdemo_rtwintro.zip`. The zip file contains the two secondary zip files, `mlrFiles.zip` and `sDirFiles.zip`.
- 6 Inspect the zip files `mlrFiles.zip` and `sDirFiles.zip`.
- 7 Relocate the zip file to a destination environment and unpack it.

Integrate Generated Code with Custom Code in External Environment

Identify required files and interfaces for calling generated code in an external build process.

Learn how to:

- Collect files required for building outside of Simulink®
- Interface with external variables and functions

For information about the example model and other examples in this series, see “Generate C Code from a Control Algorithm for an Embedded System”.

Collect and Build Required Data and Files

The code that Embedded Coder® generates requires support files that are provided by MathWorks®. To relocate the generated code to another development environment, such as a dedicated build system, you must also relocate these support files. You can package all of these files in a zip file by using the `packNGo` utility. This utility finds and packages the files that you need to build an executable image. The utility uses tools for customizing the build process after code generation, which include a `buildinfo_data` structure, as well as a `packNGo` function. These files include external files that you identify in the **Code Generation > Custom Code** pane in the Model Configuration Parameters dialog box. The utility saves the `buildinfo` MAT-file in the `model_ert_rtw` folder.

Open the example model, `rtwdemo_PCG_Eval_P5`.

This model is configured to run `packNGo` after code generation.

Generate code from the entire model.

To generate the zip file manually:

- 1 Load the file `buildInfo.mat` (located in the `rtwdemo_PCG_Eval_P5_ert_rtw` subfolder).
- 2 At the command prompt, enter the command `packNGo(buildInfo)`.

The number of files in the zip file depends on the version of Embedded Coder® and on the configuration of the model that you use. The compiler does not require all of the files in the zip file. The compiled executable size (RAM/ROM) depends on the link process. The linker likely includes only the object files that are necessary.

Integrating the Generated Code into an Existing System

This example shows how to integrate the generated code into an existing code base. The example uses the Eclipse™ IDE and the Cygwin™/gcc compiler. The required integration tasks are common to all integration environments.

Overview of Integration Environment

A full embedded controls system consists of multiple hardware and software components. Control algorithms are just one type of component. Other components can be:

- An operating system (OS)
- A scheduling layer
- Physical hardware I/O
- Low-level hardware device drivers

Typically, you do not use the generated code in these components. Instead, the generated code includes interfaces that connect with these components. MathWorks® provides hardware interface block libraries for many common embedded controllers. For examples, see the Embedded Targets block library.

This example provides files to show how you can build a full system. The main file is `example_main.c`, which contains a simple main function that performs only basic actions to exercise the code.

View `example_main.c`.

```

int_T main(void)
{
    /* Initialize model */
    rt_Pos_Command_Arbitration_Init(); /* Set up the data structures for chart*/
    rtwdemo_PCG__Define_Throt_Param(); /* SubSystem: '<Root>/Define_Throt_Param' */
    defineImportData();                /* Defines the memory and values of inputs */

    do /* This is the "Schedule" loop.
        Functions would be called based on a scheduling algorithm */
    {
        /* HARDWARE I/O */

        /* Call control algorithms */
        PI_Cntrl_Reusable((*pos_rqst),fbk_1,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_1,
                        &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_1);
        PI_Cntrl_Reusable((*pos_rqst),fbk_2,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_2,
                        &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_2);
        pos_cmd_one = rtwdemo_PCG_Eval_P5_B.PI_ctrl_1.Saturation1;
        pos_cmd_two = rtwdemo_PCG_Eval_P5_B.PI_ctrl_2.Saturation1;

        rtwdemo_Pos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two,
                                        &rtwdemo_PCG_Eval_P5_B.sf_Pos_Command_Arbitration);
    }
}

```

The file:

- Defines function interfaces (function prototypes)
- Includes files that declare external data
- Defines extern data
- Initializes data
- Calls simulated hardware
- Calls algorithmic functions

The order of function execution matches the order of subsystem execution in the test harness model and in `rtwdemo_PCG_Eval_P5.h`. If you change the order of execution in `example_main.c`, results that the executable image produces differ from simulation results.

Match System Interfaces

Integration requires matching both the *Data* and *Function* interfaces of the generated code and the existing system code. In this example, the `example_main.c` file imports and exports the data through `#include` statements and `extern` declarations. The file also calls the functions from the generated code.

Connect Input Data

The system has three input signals: `pos_rqst`, `fbk_1`, and `fbk_2`. The generated code accesses the two feedback signals through direct reference to imported global variables (storage class `ImportedExtern`). The code accesses the position signal through an imported pointer (storage class `ImportedExternPointer`).

The handwritten file `defineImportedData.c` defines the variables and the pointer. The generated code does not define the variables and the pointer because the handwritten code defines them. Instead, the generated code declares the imported data (`extern`) in the file `rtwdemo_PCG_Eval_P5_Private.h`. In a real system, the data typically come from other software components or from hardware devices.

View `defineImportedData.c`.

```
/* Define imported data */
#include "rtwtypes.h"
#include "defineImportedData.h"

real_T fbk_1;
real_T fbk_2;
real_T dummy_pos_value = 10.0;
real_T *pos_rqst;
void defineImportData(void)
{
    pos_rqst = &dummy_pos_value;
}
```

View `rtwdemo_PCG_Eval_P5_Private.h`.

```

/* Imported (extern) block signals */
extern real_T fbk_1;          /* '<Root>/fbk_1' */
extern real_T fbk_2;          /* '<Root>/fbk_2' */

/* Imported (extern) pointer block signals */
extern real_T *pos_rqst;     /* '<Root>/pos_rqst' */

```

Connect Output Data

In this example, you do not access the output data of the system. The example “Test Generated Code” shows how you can save the output data to a standard log file. You can access the output data by referring to the file `rtwdemo_PCG_Eval_P5.h`.

View `rtwdemo_PCG_Eval_P5.h`.

Access Additional Data

The generated code contains several structures that store commonly used data including:

- Block state values (integrator, transfer functions)
- Local parameters
- Time

The table lists the common data structures. Depending on the configuration of the model, some or all of these structures appear in the generated code. The data is declared in the file `rtwdemo_PCG_Eval_P5.h`, but in this example, you do not access this data.

Data Type	Data Name	Data Purpose
Constants	model_cP	Constant parameters
Constants	model_cB	Constant block I/O
Output	model_U	Root and atomic subsystem input
Output	model_Y	Root and atomic subsystem output
Internal data	model_B	Value of block output
Internal data	model_D	State information vectors
Internal data	model_M	Time and other system level data
Internal data	model_Zero	Zero-crossings
Parameters	model_P	Parameters

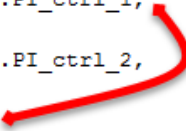
Match Function Call Interfaces

By default, functions that the code generator generates have a `void Func(void)` interface. If you configure the model or atomic subsystem to generate reentrant code,

the code generator creates a more complex function prototype. In this example, the `example_main` function calls the generated functions with the correct input arguments.

```
PI_Cntrl_Reusable((*pos_rqst),fbk_1,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_1,
                 &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_1);
PI_Cntrl_Reusable((*pos_rqst),fbk_2,&rtwdemo_PCG_Eval_P5_B.PI_ctrl_2,
                 &rtwdemo_PCG_Eval_P5_DWork.PI_ctrl_2);
pos_cmd_one = rtwdemo_PCG_Eval_P5_B.PI_ctrl_1.Saturation1;
pos_cmd_two = rtwdemo_PCG_Eval_P5_B.PI_ctrl_2.Saturation1;

rtwdemo_Pos_Command_Arbitration(pos_cmd_one, &Throt_Param, pos_cmd_two,
                                &rtwdemo_PCG_Eval_P5_B.sf_Pos_Command_Arbitration);
```



Calls to the function `PI_Cntrl_Reusable` use a mixture of separate, unstructured global variables and Simulink® Coder™ data structures. The handwritten code defines these variables. The structure types are defined in `rtwdemo_PCG_Eval_P5.h`.

Build Project in Eclipse™ Environment

This example uses the Eclipse™ IDE and the Cygwin™ GCC debugger to build the embedded system. The example provides installation files for both programs. Software components and versions numbers are:

- Eclipse™ SDK 3.2
- Eclipse™ CDT 3.3
- Cygwin™/GCC 3.4.4-1
- Cygwin™/GDB 20060706-2

To install and use Eclipse™ and GCC, see “Install and Use Cygwin and Eclipse”.

You can install the files for this example automatically by clicking this [hyperlink](#).

Set up the build folder.

Alternatively, to install the files manually:

- 1 Create a build folder (`Eclipse_Build_P5`).
- 2 Unzip the file `rtwdemo_PCG_Eval_P5.zip` into the build folder.
- 3 Delete the files `rtwdemo_PCG_Eval_P5.c`, `ert_main.c` and `rt_logging.c`, which are replaced by `example_main.c`.

Note: Before continuing to the next example in the series, if you have not generated code for the model or if the zip file does not exist, complete the steps in **Collect and Build Required Data and Files**.

You can use the Eclipse™ debugger to step through and evaluate the execution behavior of the generated C code. See the example “Install and Use Cygwin and Eclipse”.

The next example in this series, “Test Generated Code”, shows how to exercise the model with input data.

packNGo Function Limitations

The following limitations apply to the `packNGo` function:

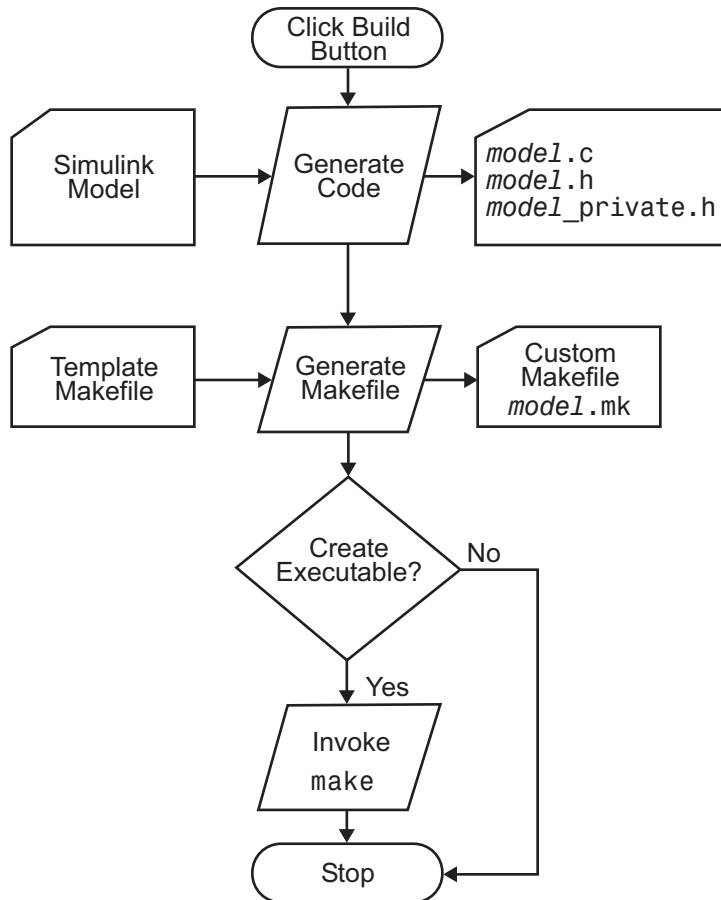
- The function operates on source files, such as `*.c`, `*.cpp`, and `*.h` files, only. The function does not support compile flags, defines, or makefiles.
- Unnecessary files might be included. The function might find additional files from source and include paths recorded in the model's build information and include them, even if they are not used.

More About

- `packNGo`
- “Configure Build Process” on page 25-13
- “Build and Run a Program” on page 25-36
- “How Executable Programs Are Built From Models” on page 25-39
- “Executable Program Generation” on page 25-63

Executable Program Generation

The following figure shows how the process for automatic program building.

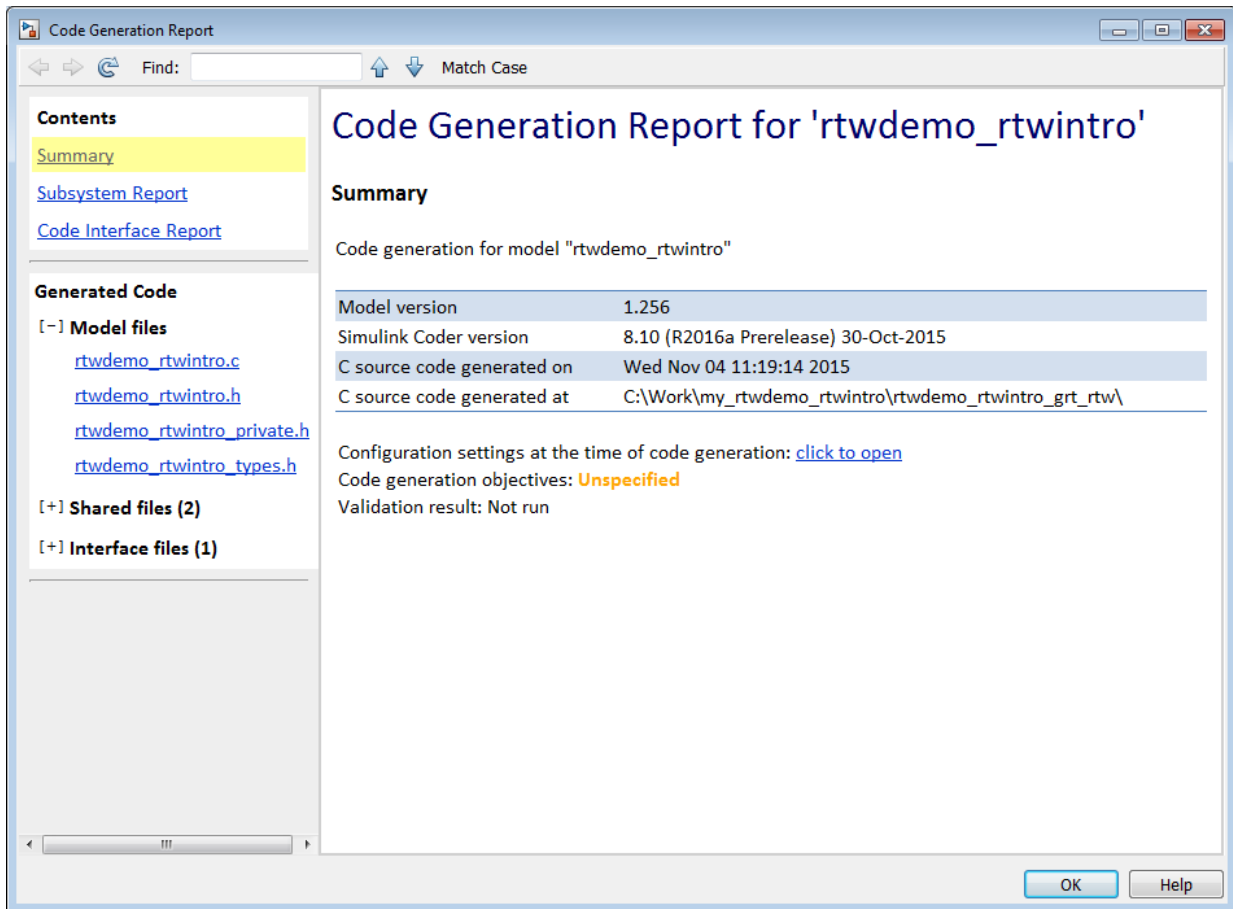


During the final stage of processing, the build process invokes the generated makefile, `model.mk`, which in turn compiles and links the generated code. On PC platforms, a batch file is created to invoke the generated makefile. The batch file sets up the environment for invoking the `make` utility and related compiler tools. To avoid recompiling C files, the `make` utility performs date checking on the dependencies between the object and C files; only out-of-date source files are compiled. Optionally, the makefile can download the resulting executable image to your target hardware.

This stage is optional, as illustrated by the control logic in the preceding figure. You might choose to omit this stage, for example, if you are targeting an embedded microcontroller or a digital signal processing (DSP) board.

To omit this stage of processing, select the **Generate code only** check box on the **Code Generation** pane of the Configuration Parameters dialog box. You can then cross-compile your code and download it to your target hardware.

If you select **Create code generation report** on the **Code Generation > Report** pane, a navigable summary of source files is produced when the model is built. The report files occupy a folder called `html` within the build folder. The following display shows an example of an HTML code generation report for a generic real-time (GRT) target.



More About

- “Configure Build Process” on page 25-13
- “Initiate the Build Process” on page 25-35
- “Build and Run a Program” on page 25-36
- “Executable Program Generation” on page 25-63

Profile Code Performance

By profiling the performance of generated code, you can help verify that the code meets performance requirements.

About Profiling Code Performance

Profiling can be especially important early in the development cycle for identifying potential architectural issues that can be more expensive to address later in the process. Profiling can also identify bottlenecks and procedural issues that indicate a need for optimization, for example, with an inner loop or inline code.

Note: If you have an Embedded Coder license, see “Code Execution Profiling” for an alternative and simpler approach based on software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations.

How to Profile Code Performance

You can profile code generated with code generation technology by using a Target Language Compiler (TLC) hook function interface.

To use the profile hook function interface:

- 1 For your target, create a TLC file that defines the following hook functions. Write the functions so that they specify profiling code. The code generator adds the hook function code to code generated for atomic systems in the model.

Function	Input Arguments	Output Type	Description
ProfilerHeaders	void	Array of header file names	Return an array of the header file names to be included in the generated code.
ProfilerTypedefs	void	typedefs	Generate code statements for profiler type definitions.
ProfilerGlobal-Data	system	Global data for the specified system	Generate code statements that declare global data.

Function	Input Arguments	Output Type	Description
ProfilerExtern-DataDecls	system	extern declarations for the specified system	Generate code statements that create global extern declarations.
ProfilerSystem-Decls	system, functionType	Declarations for the specified system for the specified functionType	Generate code for required variable declarations within the scope of an atomic subsystem Output, Update, OutputUpdate, or Derivatives function.
ProfilerSystem-Start	system, functionType	Profiler start commands for the specified system and functionType	Generate code that starts the profiler within the scope of an atomic subsystem Output, Update, OutputUpdate, or Derivatives function.
ProfilerSystem-Finish	system, functionType	Profiler end commands for the specified system and functionType	Generate code that stops the profiler within the scope of an atomic subsystem's Output, Update, OutputUpdate, or Derivatives function.
ProfilerSystem-Terminate	system	Profiler termination code for the specified system	Generate code that terminates profiling (and possibly reports results) for an atomic subsystem.

For an example TLC file, see `matlabroot/toolbox/rtw/rtwdemos/rtwdemo_profile_hook.tlc`.

- 2 In your `target.tlc` file, define the following global variables.

Define...	To Be...
ProfileGenCode	TLC_TRUE or 1 to turn on profiling (TLC_FALSE or 0 to turn off profiling)
ProfilerTLC	The name of the TLC file that you created in step 1

A quick way to define global variables is to define the parameters with the `-a` option. You can do this by using the `set_param` command to set the model configuration parameter `TLCOptions`. For example,

```
>> set_param(gcs, 'TLCOptions', '-aProfileGenCode=1 -aProfilerTLC="rtwdemo_profile_hook.tlc"')
```

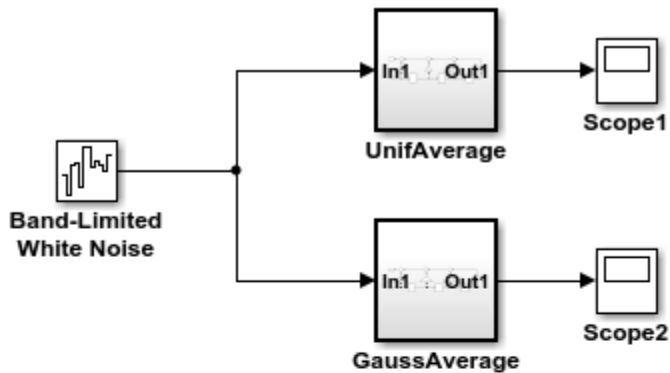
- 3 Consider setting configuration parameters for generating a code generation report. You can then examine the profiling code in the context of the code generated for the model.
- 4 Build the model. The build process embeds the profiling code in the hook function locations in the generated code for the model.
- 5 Run the generated executable file. In the MATLAB Command Window, enter `!model -name`. You see the profiling report you programmed in the profiling TLC file that you created. For example, a profile report might list the number of calls made to each system in a model and the number of CPU cycles spent in each system.

For an example, see “Run Profiling Hooks for Generated Code” on page 25-69. For details on programming a `.tlc` file and defining TLC configuration variables, see “Target Language Compiler”.

Embed Profiling Commands In Generated Code

This model shows the hooks provided by Simulink® Coder™ to profile generated code. In this model, the subsystems that perform averaging are profiled. Use the blue buttons in the model to generate and inspect code. The code contains profiling commands. To learn more about how profiler code was added for this example, look at the file `rtwdemo_profile_hook.tlc`.

```
open_system('rtwdemo_profile')
```



This model shows the hooks provided by Simulink Coder to profile generated code. In this model, the subsystems that perform averaging are profiled. Use the blue buttons below to generate code for this model and inspect the generated code. The code contains profiling commands. To learn more about how profiler code was added for this example, look at the file `rtwdemo_profile_hook.tlc`.

`matlabroot\toolbox\rtw\rtwdemos\rtwdemo_profile_hook.tlc`

**Generate Code Using
Simulink Coder
(double-click)**

**Generate Code Using
Embedded Coder
(double-click)**

**View documentation
(double-click)**

Copyright 1994-2012 The MathWorks, Inc.

Run Profiling Hooks for Generated Code

The example model `rtwdemo_profile` shows how to use special hooks provided by the code generator to add profiling code snippets to the generated code for your models.

The basis for this example is the Target Language Compiler (TLC) hook function interface described in “How to Profile Code Performance” on page 25-66. The hooks

allow you to add profiling code snippets within nonvirtual (atomic) systems in the model, which allow you to profile those atomic systems. For this example, the file `matlabroot/toolbox/rtw/rtwdemos/rtwdemo_profile_hook.tlc` implements the required profiling functions. You can inspect this file for more information on each of the profiling functions.

Run the example as follows:

- 1 Open the model `rtwdemo_profile`; for example, you can enter the command `rtwdemo_profile` in the MATLAB Command Window.
- 2 Change your working folder to one for which you have write permission.
- 3 Double-click one of the Generate Code blocks. This configures the selected code generation target options with the profiling hooks and builds the model. When the build process is completed, a window opens with a display of the generated code. Browse through the code to see the profile-specific C code that has been generated.
- 4 Enter `!rtwdemo_profile` in the MATLAB Command Window to run the generated executable file. This command displays a textual report of the number of calls made to each system in the example model and the number of CPU cycles that were spent in each system.

Profiling Limitation

The TLC hook function interface for profiling code performance does not support the S-function target (`rtwsfcn.tlc`).

More About

- “Run Profiling Hooks for Generated Code” on page 25-69
- “Build and Run a Program” on page 25-36
- “Code Execution Profiling”

Host/Target Communication in Simulink Coder

Set Up and Use Host/Target Communication Channel

External mode allows two separate systems, a *host* and a *target*, to communicate. The host is the computer where the MATLAB and Simulink environments execute. The target is the computer where the executable created by the code generation and build process runs.

The host (the Simulink environment) transmits messages requesting the target to accept parameter changes or to upload signal data. The target responds by executing the request. External mode communication is based on a *client/server* architecture, in which the Simulink environment is the client and the target is the server.

In this section...

“What You Can Do With a Host/Target Communication Channel” on page 26-2

“Set Up an External Mode Communication Channel” on page 26-3

“Configure and Use External Mode” on page 26-14

“External Mode Compatible Blocks and Subsystems” on page 26-34

“External Mode Communication” on page 26-37

“Choose Communication Protocol for Client and Server” on page 26-40

“Use External Mode Programmatically” on page 26-49

“Animate Stateflow Charts in External Mode” on page 26-54

“External Mode Limitations” on page 26-55

What You Can Do With a Host/Target Communication Channel

You can use a host/target communication channel to:

- Modify, or *tune*, block parameters in real time. In external mode, whenever you change parameters in the block diagram, the Simulink engine downloads them to the executing target program. You can tune your program parameters without recompiling.
- View and log block outputs in many types of blocks and subsystems. You can monitor and store signal data from the executing target program, without writing special interface code. You can define the conditions under which data is uploaded from target to host. For example, data uploading can be triggered by a selected signal crossing zero in a positive direction. Alternatively, you can manually trigger data uploading.

External mode establishes a communication channel between the Simulink engine and generated code. The channel's low-level *transport layer* handles the physical transmission of messages. The Simulink engine and the generated model code are independent of this layer. The transport layer and the code directly interfacing to it are isolated in separate modules that format, transmit, and receive messages and data packets.

This design allows for different targets to use different transport layers. For example:

- ERT, GRT, and RSim targets support external mode host/target communication by using TCP/IP and serial (RS-232) communication.
- The Simulink Real-Time product uses a customized transport layer.
- The Simulink Desktop Real-Time product uses shared memory communication.
- Target hardware platforms supported by Simulink use serial or TCP/IP communication.

For an example of using external mode, see “Set Up an External Mode Communication Channel” on page 26-3.

Set Up an External Mode Communication Channel

- “External Mode Communication Channel Setup” on page 26-3
- “Set Up the Model” on page 26-4
- “Build the Target Executable” on page 26-6
- “Run the External Mode Target Program” on page 26-8
- “Tune Parameters” on page 26-12

External Mode Communication Channel Setup

External mode is a very useful environment for rapid prototyping. This example consists of four parts, each of which depends on completion of the preceding ones, in order. The four parts correspond to the steps that you follow in simulating, building, running, and tuning an actual real-time application.

- 1 Set up the model.
- 2 Build the target executable.
- 3 Run the external mode target program.
- 4 Tune parameters.

The example uses the GRT target. It does not require hardware other than the computer on which you run the Simulink and Simulink Coder software. The generated executable in this example runs on the host computer in a separate process from MATLAB and Simulink.

Set Up the Model

In this part of the example, you create a simple model, `ex_extModeExample`. You also create a folder called `ext_mode_example` to store the model and the generated executable.

To create the folder and the model:

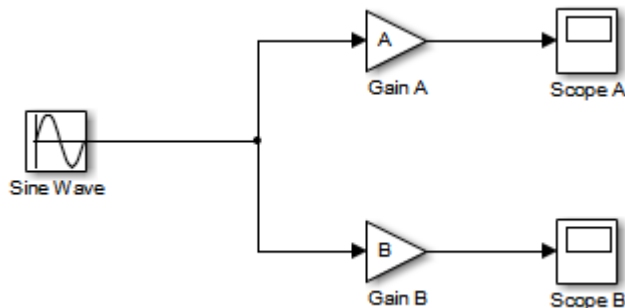
- 1 From the MATLAB command line, type:

```
mkdir ext_mode_example
```

- 2 Make `ext_mode_example` your working folder:

```
cd ext_mode_example
```

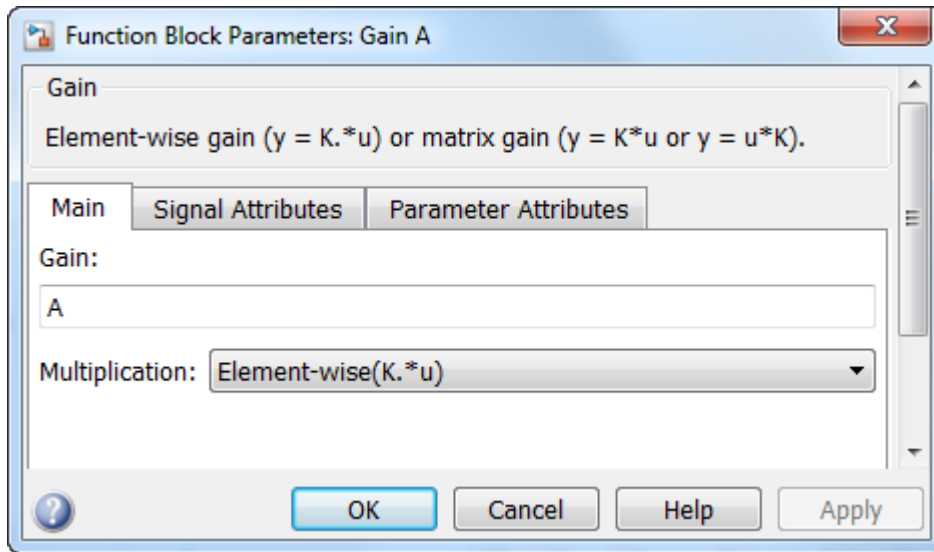
- 3 Create a model in Simulink with a Sine Wave block for the input signal, two Gain blocks in parallel, and two Scope blocks. Be sure to label the Gain and Scope blocks as shown below.



- 4 Define and assign two MATLAB workspace variables, A and B, as follows:

```
A = 2;  
B = 3;
```

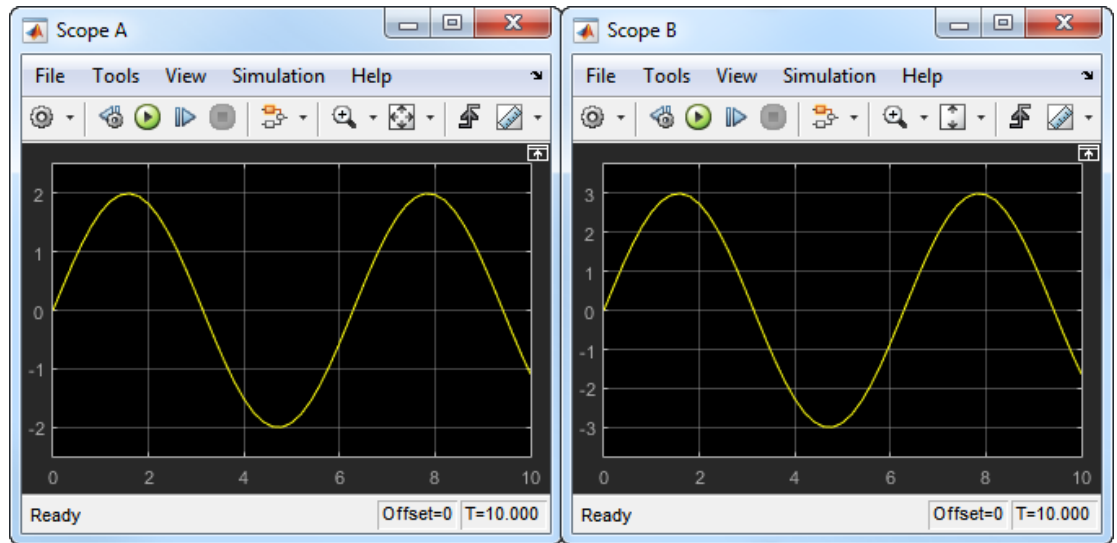
- 5 Open Gain block A and set its **Gain** parameter to the variable A.



- 6 Open Gain block B and set its **Gain** parameter to the variable B.

When the target program is built and connected to Simulink in external mode, you can download new gain values to the executing target program. To do this, you can assign new values to workspace variables A and B, or edit the values in the block parameters dialog box. For more information, see “Tune Parameters” on page 26-12.

- 7 Verify operation of the model. Open the Scope blocks and run the model. When A = 2 and B = 3, the output appears as shown below.



- 8 Save the model as `ex_extModeExample`.

Build the Target Executable

Set up the model and code generation parameters required for an external mode compatible target program. Then, generate code and build the target executable.

- 1 Open the Configuration Parameters dialog box by selecting **Simulation > Model Configuration Parameters**.
- 2 Select the **Solver** pane.
- 3 In the **Solver options** subpane:
 - a In the **Type** field, select **Fixed-step**.
 - b In the **Solver** field, select **discrete (no continuous states)**.
 - c Open **Additional options**. In the **Fixed-step size** field, specify **0.1**. (Otherwise, when you generate code, the Simulink Coder build process posts a warning and supplies a value.)

Leave **Start time** set to the default value of **0.0**.

Click **Apply**.

- 4 Select the **Data Import/Export** pane, and clear the **Time** and **Output** check boxes. In this example, data is not logged to the workspace or to a MAT-file. Click **Apply**.
- 5 Select the **Optimization > Signals and Parameters** pane. Make sure that **Default parameter behavior** is set to Tunable. Inlined parameters are not part of this example. If you have made changes, click **Apply**.
- 6 Select the **Code Generation** pane. By default, the generic real-time (GRT) target is selected.
- 7 Select the **Code Generation > Interface** pane. In the **Data exchange interface** section, select **External mode**. This selection enables generation of external mode support code and displays additional external mode configuration parameters.
- 8 In the **External mode configuration** section, make sure that the default value `tcpip` is selected for the **Transport layer** parameter.

External mode

External mode configuration

Transport layer: MEX-file name: `ext_comm`

MEX-file arguments:

Static memory allocation

External mode supports communication via TCP/IP, serial, and custom transport protocols. The **MEX-file name** field specifies the name of a MEX-file that implements host and target communication on the host side. The default for TCP/IP is `ext_comm`, a MEX-file provided with the Simulink Coder software. You can override this default by supplying other files. If you need to support other transport layers, see “Create a Transport Layer for External Communication” on page 36-8.

The **MEX-file arguments** field lets you specify arguments, such as a TCP/IP server port number, to be passed to the external interface program. These arguments are specific to the external interface that you are using. For information on setting these arguments, see “MEX-File Optional Arguments for TCP/IP Transport” on page 26-43 and “MEX-File Optional Arguments for Serial Transport” on page 26-45 in the Simulink Coder documentation.

This example uses the default arguments. Leave the **MEX-file arguments** field blank.

- 9 Click **Apply** to save the external mode settings.
- 10 Save the model.
- 11 Select the **Code Generation** pane. Make sure that **Generate code only** is cleared, and then, in the model window, press **Ctrl+B** to generate code and create the target program. The software creates the `ex_extModeExample` target executable in your working folder.

Run the External Mode Target Program

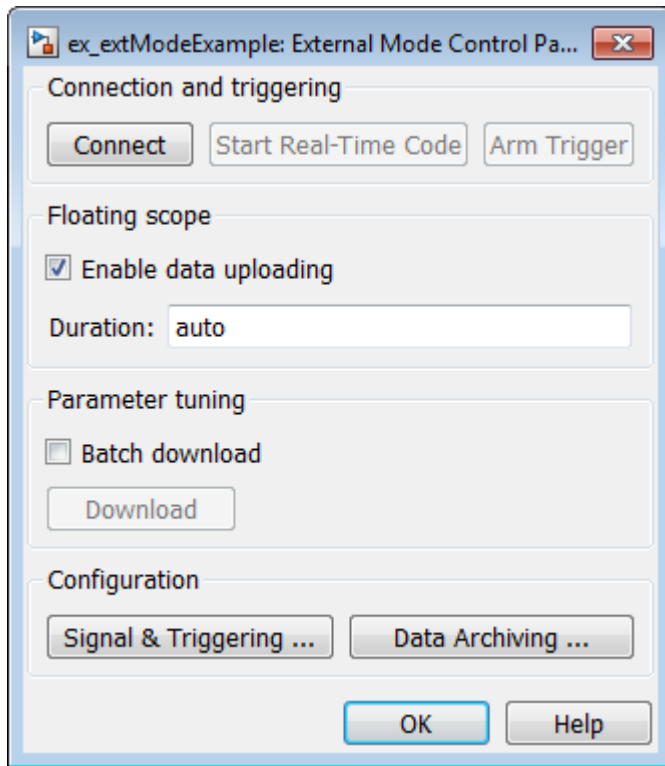
You now run the `ex_extModeExample` target executable and use Simulink as an interactive front end to the running target program. The executable file is in your working folder. Run the target program and establish communication between Simulink and the target.

Note: An external mode program like `ex_extModeExample` is a host-based executable. Its execution is not tied to a real-time operating system (RTOS) or a periodic timer interrupt, and it does not run in real time. The program just runs as fast as possible, and the time units it counts off are simulated time units that do not correspond to time in the world outside the program.

The External Signal & Triggering dialog box (accessed from the External Mode Control Panel) displays a list of blocks in your model that support external mode signal monitoring and logging. In the dialog box, you can configure the signals that are viewed, how they are acquired, and how they are displayed.

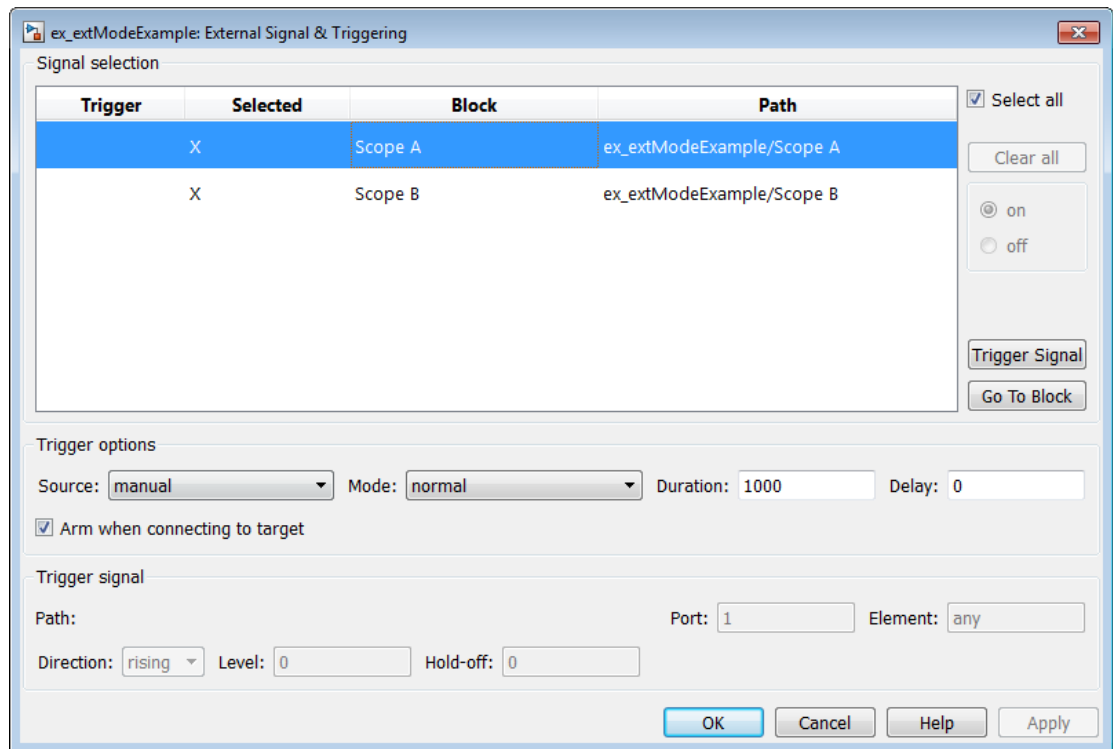
In this example, you observe and use the default settings of the External Signal & Triggering dialog box.

- 1 From the **Code** menu of the model diagram, select **External Mode Control Panel**. This control panel is where you configure signal monitoring and data archiving. You can also connect to the target program, and start and stop execution of the model code.



- You use the top row of buttons after the target program has started.
- The **Signal & Triggering** button opens the External Signal & Triggering dialog box. Use this dialog box to select the signals that are collected from the target system and viewed in external mode. You can also select a signal that triggers uploading of data when certain signal conditions are met, and define the triggering conditions.
- The **Data Archiving** button opens the Enable Data Archiving dialog box. Use data archiving to save data sets generated by the target program for future analysis. This example does not use data archiving. See “Configure Host Archiving of Target Program Signal Data” on page 26-31 for more information.

- Click the **Signal & Triggering** button to open the External Signal & Triggering dialog box. The default configuration selects all signals for monitoring and sets signal monitoring to begin as soon as the host and target programs have connected.



- Check that the External Signal & Triggering dialog box is set to the following defaults:

- Select all** check box is selected. Signals in the **Signal selection** list are marked with an X in the **Selected** column.
- Under **Trigger options**:
 - Source**: manual
 - Mode**: normal
 - Duration**: 1000
 - Delay**: 0

- **Arm when connecting to target:** selected

Click **OK** to close the External Signal & Triggering dialog box, and then close the External Mode Control Panel.

For descriptions of the External Signal & Triggering dialog box parameters, see “Configure Host Monitoring of Target Program Signal Data” on page 26-23.

Note: At the conclusion of this example, when Simulink is connected to a running target executable, you can modify signal and triggering options without disconnecting from the target. For more information, see “Modify Signal and Triggering Options While Connected” on page 26-30.

- 4 To run the target program, open an operating system command window (on UNIX systems, a terminal emulator window). At the command prompt, use `cd` to navigate to the `ext_mode_example` folder to which you generated the target program executable.

Enter the following command:

```
ex_extModeExample -tf inf -w
```

Note Alternatively, you can run the target program from the MATLAB Command Window, using the following syntax.

```
!ex_extModeExample -tf inf -w &
```

The target program begins execution, and enters a wait state.

The `-tf` switch overrides the stop time set for the model in Simulink. The `inf` value directs the model to run indefinitely. The model code runs until the target program receives a stop message from Simulink.

The `-w` switch instructs the target program to enter a wait state until it receives a **Start Real-Time Code** message from the host. If you want to view data from time step 0 of the target program execution, or if you want to modify parameters before the target program begins execution of model code, this switch is required.

- 5 Open the Scope blocks in the model. Signals are not visible on the scopes. When you connect Simulink to the target program and begin model execution, the signals generated by the target program become visible on the scope displays.
- 6 Before communication between the model and the target program can begin, the model must be in external mode. To enable external mode, from the **Simulation > Mode** menu, select **External**.
- 7 Reopen the External Mode Control Panel (found in the **Code** menu) and click **Connect**. This action initiates a handshake between Simulink and the target program. When Simulink and the target are connected, the **Start Real-Time Code** button becomes enabled, and the label of the **Connect** button changes to **Disconnect**.
- 8 Click **Start Real-Time Code**. The outputs of Gain blocks A and B are displayed on the two scopes in your model.

You have established communication between Simulink and the running target program. You can now tune block parameters in Simulink and observe the effects the parameter changes have on the target program.

Tune Parameters

You can change the gain factor of either Gain block by assigning a new value to the variable **A** or **B** in the MATLAB workspace. When you change block parameter values in the workspace during a simulation, you must explicitly update the block diagram with these changes. When you update the block diagram, the new values are downloaded to the target program.

Under certain conditions, you can also tune the expressions that you use to specify block parameter values. To change an expression during simulation, open the block dialog box.

- 1 At the command prompt, assign new values to both variables, for example:

```
A = 0.5;  
B = 3.5;
```
- 2 Open the `ex_extModeExample` model window. From the **Simulation** menu, select **Update Diagram**. As soon as Simulink has updated the block parameters, the new gain values are downloaded to the target program, and the scopes are changed because of the gain change.
- 3 In the Sine Wave block dialog box, set **Amplitude** to **0.5**. Click **Apply** or **OK**.

When you click **Apply** or **OK**, the simulation downloads the new block parameter value to the target program. The Scope block displays the change to reflect the new amplitude value.

- 4 To simultaneously disconnect host/target communication and end execution of the target program, from the **Simulation** menu, select **Stop**. Alternatively, in the External Mode Control Panel, click **Stop Real-Time Code**.

You cannot change the sample time of the Sine Wave block during simulation. Block sample times are part of the structural definition of the model and are part of the generated code. Therefore, if you want to change a block sample time, you must stop the external mode simulation, reset the sample time of the block, and rebuild the executable.

Block parameter tunability during external mode simulation depends on the way that the generated code represents block parameters.

For example, in the Gain A block dialog box, you cannot change the expression A in the **Gain** parameter during simulation. Instead, you must change the value of the variable A in the base workspace. You cannot change the expression because the generated code does not allocate storage in memory for the **Gain** parameter. Instead, the code creates a field A in a structure:

```
/* Parameters (auto storage) */
struct P_ex_extModeExample_T_ {
    real_T A;                /* Variable: A
                           */
    real_T B;                /* Variable: B
                           */
    real_T SineWave_Amp;     /* Expression: 1
                           */
    real_T SineWave_Bias;    /* Expression: 0
                           */
    real_T SineWave_Freq;    /* Expression: 1
                           */
    real_T SineWave_Phase;   /* Expression: 0
                           */
};
```

The generated code algorithm uses that field in the code that represents the block Gain A. In this case, the global structure variable `ex_extModeExample_P` uses the type `P_ex_extModeExample_T_`:

```
ex_extModeExample_B.GainA = ex_extModeExample_P.A * rtb_SineWave;
```

When you change the value of **A** in the base workspace, the simulation downloads the new value to the field **A** in the target program.

However, you can change the expressions in the Sine Wave block parameters during simulation. You can change these expressions because the generated code creates a field in the global structure `ex_extModeExample_P` to represent each parameter in the block. When you change an expression in the block dialog box, the simulation first evaluates the new expression. The simulation then downloads the resulting numeric value to the corresponding structure field in the target program.

For more information about controlling parameter representation in the generated code, see “Block Parameter Representation in the Generated Code” on page 14-43.

Configure and Use External Mode

- “Configure External Mode Options for Code Generation” on page 26-14
- “Target Interfacing” on page 26-16
- “Control Host and Target Execution” on page 26-18
- “Control External Mode Operations” on page 26-19
- “Configure Host Monitoring of Target Program Signal Data” on page 26-23
- “Configure Host Archiving of Target Program Signal Data” on page 26-31

Configure External Mode Options for Code Generation

The list of targets and products that support external mode includes the ERT, GRT, and RSim targets, the Simulink Desktop Real-Time product, and Simulink target hardware platforms. Code generation targets that support external mode provide a set of external mode options in the Configuration Parameters dialog box, on the **Code Generation > Interface** pane or their respective target pane.

For targets that support the ability to build target code, connect to the target, and run an application in external mode by clicking **Run** in the model window toolbar, note the following:

- You select external mode from the model window by clicking **Simulation > Mode > External**.
- External mode parameters continue to appear for the target in the Configuration Parameters dialog box.
- By default, the code generator produces code that is *not* set up for external mode if you build the code by using one of following methods:

- Press **Ctrl+B**
- At the MATLAB command line, enter `rtwbuild`

The following figure shows external mode parameters from the GRT and ERT target views of the **Code Generation > Interface** pane.

External mode

External mode configuration

Transport layer: tcpip ▼ MEX-file name: `ext_comm`

MEX-file arguments:

Static memory allocation

Note The Simulink Real-Time product also uses external mode communication. External mode in the Simulink Real-Time product is always on, and does not have interface options.

The **Data exchange interface** section in the **Code Generation > Interface** pane includes the following external mode parameters:

- **External mode** option: Includes the external mode data exchange interface in the generated C code.

When you select **External mode**, the following parameters appear:

- **Transport layer** menu: Identifies the messaging protocol for host/target communication; typically, the choices are `tcpip` and `serial`.

The default is `tcpip`. When you select a protocol, the MEX-file name that implements the protocol is shown to the right of the menu.

- **MEX-file arguments** text field: Optionally enter a list of arguments to be passed to the transport layer MEX-file for communicating with executing targets. The arguments vary according to the protocol that you use.

For more information on the transport options, see “Target Interfacing” on page 26-16 and “Choose Communication Protocol for Client and Server” on page 26-40.

- **Static memory allocation** check box: Controls how memory is allocated for external mode communication buffers in the target. Selecting this option enables the **Static memory buffer size** parameter.
- **Static memory buffer size** text field: Number of bytes to preallocate for external mode communication buffers in the target.

Note Selecting **External mode** does not cause the Simulink model to operate in external mode (see “Control Host and Target Execution” on page 26-18). The **External mode** option instruments the code generated for the target to support external mode.

The **Static memory allocation** check box (for GRT and ERT targets) directs the Simulink Coder software to generate code for external mode that uses only static memory allocation (“malloc-free” code). Selecting **Static memory allocation** enables the **Static memory buffer size** edit field, which you use to specify the size of the static memory buffer used by external mode. The default value is 1,000,000 bytes. If you enter too small a value for your application, external mode issues an out-of-memory error when it tries to allocate more memory than you allowed. In such cases, increase the value in the **Static memory buffer size** field and regenerate the code.

To determine how much memory to allocate, enable verbose mode on the target (by including `OPTS=" -DVERBOSE "` on the `make` command line). As it executes, external mode displays the amount of memory it tries to allocate and the amount of memory available to it each time it attempts an allocation. If an allocation fails, you can use this console log to adjust the size in the **Static memory buffer size** field.

Note When you create an ERT target, external mode can generate pure integer code. Select pure integer code by clearing the **Support floating-point numbers** option on the **Code Generation > Interface** pane of the Configuration Parameters dialog box. Clearing this option makes the code, including external mode support code, free of doubles and floats. For more information, see “Model Configuration Parameters: Code Generation Interface”.

Target Interfacing

The Simulink Coder product lets you implement client and server transport for external mode using either TCP/IP or serial protocols. If your target system supports TCP/IP, you can use the socket-based external mode implementation provided by the Simulink Coder

product with the generated code. Otherwise, use or customize the serial transport layer option.

A low-level *transport layer* handles physical transmission of messages. Both the Simulink engine and the model code are independent of this layer. Both the transport layer and code directly interfacing to the transport layer are isolated in separate modules that format, transmit, and receive messages and data packets.

You specify the transport mechanism using the **Transport layer** menu in the **External mode configuration** section of the **Code Generation > Interface** pane of the Configuration Parameters dialog box.

External mode

External mode configuration

Transport layer: MEX-file name: ext_comm

MEX-file arguments:

Static memory allocation

To the right of the **Transport layer** menu, **MEX-file name** displays the name of an external interface MEX-file. This MEX-file implements host/target communication for the selected external mode transport layer. The default is `ext_comm`, the TCP/IP-based external interface file for use with the GRT, ERT, and RSim targets. If you select the serial transport option, the MEX-file name `ext_serial_win32_com` is displayed in this location.

Note: Custom or third-party targets can use a custom transport layer and a different external interface MEX-file. For more information, see “Create a Transport Layer for External Communication” on page 36-8. For more information on specifying a TCP/IP or serial transport layer for a custom target, see “Using the TCP/IP Implementation” on page 26-41 or “Using the Serial Implementation” on page 26-44.

In the **MEX-file arguments** edit field, you can optionally specify arguments that are passed to the external mode interface MEX-file for communicating with executing targets. The meaning of the MEX-file arguments depends on the MEX-file implementation.

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

- Network name of your target (for example, 'myPuter' or '148.27.151.12')
- Verbosity level (0 for no information or 1 for detailed information)
- TCP/IP server port number (an integer value between 256 and 65535, with a default of 17725)



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

- Verbosity level (0 for no information or 1 for detailed information)
- Serial port ID (for example, 1 for COM1, and so on)
- Baud rate (selected from the set 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, and 115200, with a default baud rate of 57600)

For more information on MEX-file transport architecture and arguments, see “Choose Communication Protocol for Client and Server” on page 26-40.

Control Host and Target Execution

Simulink software provides multiple ways to control external mode host and target execution. The following table lists common steps in the external mode workflow and the ways in which they can be initiated.

External Mode Action	Toolbar Control	Menu Control	External Mode Control Panel Button
Set the simulation mode of your model to external mode	From the simulation mode drop-down list, select External	Simulation > Mode > External	Connect (if the model simulation mode is not already set, sets the simulation mode to external mode)
Connect your model to a waiting or running target program	Connect to Target button 	Simulation > Connect to Target	Connect
Start running real-time code in the target environment	Run button	Simulation > Run (keyboard shortcut Ctrl +T)	Start Real-Time Code
Disconnect your model from the target	Disconnect from Target button 	Simulation >	Disconnect

External Mode Action	Toolbar Control	Menu Control	External Mode Control Panel Button
environment (does not halt running real-time code)		Disconnect from Target	
Stop target program execution and disconnect your model from the target environment	Stop button	Simulation > Stop (keyboard shortcut Ctrl+Shift+T)	Stop Real-Time Code

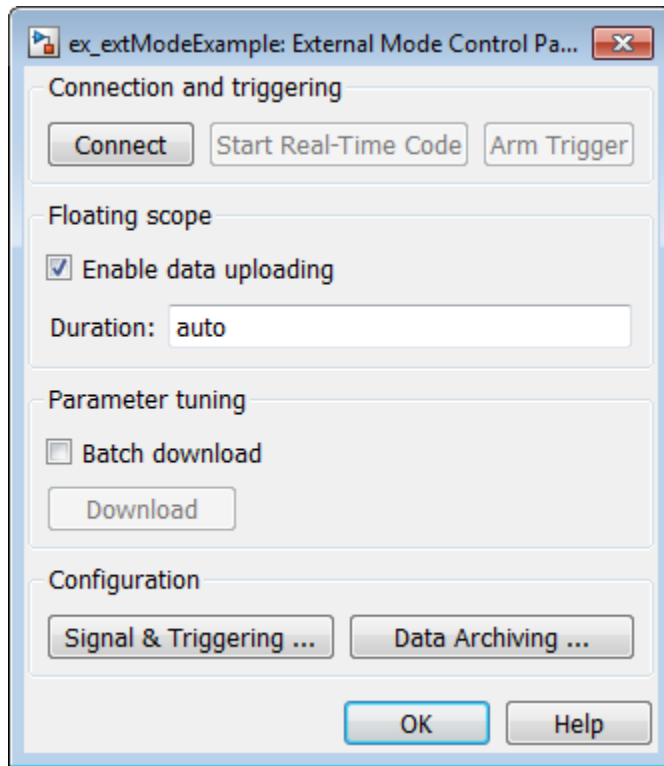
Note Setting the simulation mode of your model to external mode affects execution only, and does *not* cause the Simulink Coder software to generate code instrumented for external mode. See “Configure External Mode Options for Code Generation” on page 26-14.

Control External Mode Operations

The External Mode Control Panel provides centralized control of external mode operations, including:

- “Connect, Start, and Stop” on page 26-20
- “Upload Target Program Signal Data to Host” on page 26-21
- “Download Parameters to Target Program” on page 26-22
- “Configure Host Monitoring of Target Program Signal Data” on page 26-23
- “Configure Host Archiving of Target Program Signal Data” on page 26-31

To open the External Mode Control Panel dialog box, in the model window, select **Code > External Mode Control Panel**.



Connect, Start, and Stop

The External Mode Control Panel performs the same connect/disconnect and start/stop functions found in the **Simulation** menu and the Simulink toolbar (see “Control Host and Target Execution” on page 26-18).

Clicking the **Connect** button connects your model to a waiting or running target program. While you are connected, the button changes to a **Disconnect** button.

Disconnect disconnects your model from the target environment, but does not halt real-time code running in the target environment.

Connect sets the model simulation mode to external mode.

Clicking the **Start Real-Time Code** button commands the target to start running real-time code. While real-time code is running in the target environment, the button changes

to a **Stop Real-Time Code** button. **Stop Real-Time Code** stops target program execution and disconnects your model from the target environment.

Upload Target Program Signal Data to Host

The External Mode Control Panel allows you to trigger and cancel data uploads to the host. The destination for the uploaded data can be a scope block, Display block, To Workspace block, or another block or subsystem listed in “External Mode Compatible Blocks and Subsystems” on page 26-34.

The **Arm Trigger** and **Cancel Trigger** buttons provide manual control of data uploading to compatible blocks or subsystems, except floating scopes. (For floating scopes, use the **Floating scope** section of the External Mode Control Panel.)

- To trigger data uploading to compatible blocks or subsystems, click the **Arm Trigger** button. The button changes to **Cancel Trigger**.
- To cancel data uploading, click the **Cancel Trigger** button. The button reverts to **Arm Trigger**.

You can trigger data uploads manually or automatically. To configure signals and triggers for data uploads, see “Configure Host Monitoring of Target Program Signal Data” on page 26-23.

A subset of external mode compatible blocks, including Scope, Time Scope, and To Workspace, allow you to log uploaded data to disk. To configure data archiving, see “Configure Host Archiving of Target Program Signal Data” on page 26-31.

The **Floating scope** section of the External Mode Control Panel controls when and for how long data is uploaded to Floating Scope blocks. When used in external mode, floating scopes:

- Do not appear in the External Signal & Triggering dialog box.
- Do not log data to external mode archiving.
- Support manual triggering only.

The **Floating scope** section contains the following parameters:

- **Enable data uploading** option, which functions as an **Arm Trigger** button for floating scopes. When the target is disconnected, the option controls whether to arm the trigger when connecting the floating scopes. When the target is connected, the option acts as a toggle button to arm or cancel the trigger.

- To trigger data uploading to floating scopes, select **Enable data uploading**.
- To cancel data uploading to floating scopes, clear **Enable data uploading**.
- **Duration** edit field, which specifies the number of base-rate steps for which external mode logs floating scopes data after a trigger event. By default, it is set to **auto**, which causes the duration value set in the External Signal & Triggering dialog box to be used. The default duration value is 1000 base rate steps.

Download Parameters to Target Program

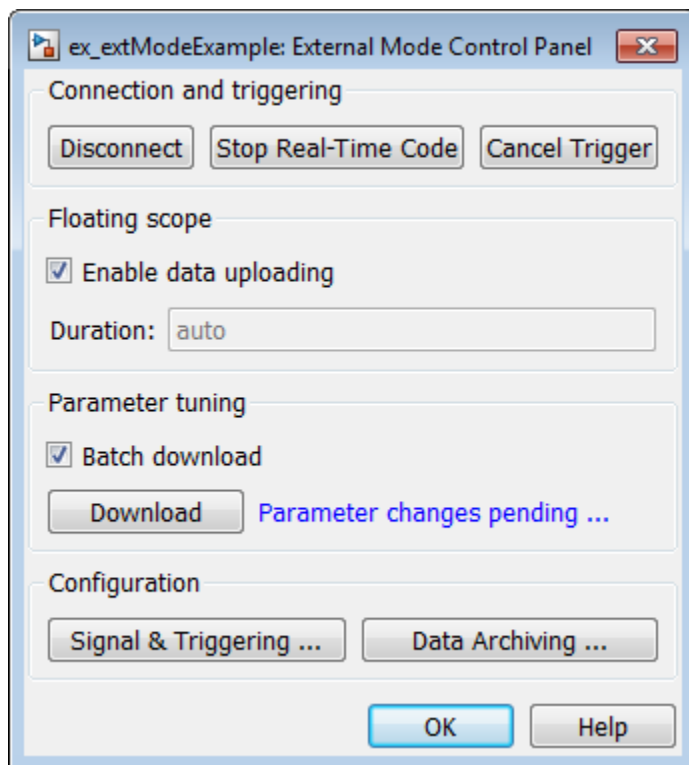
The **Batch download** option on the External Mode Control Panel enables or disables batch parameter changes.

By default, batch download is disabled. If batch download is disabled, when you click **OK** or **Apply**, changes made directly to block parameters by editing block parameter dialog boxes are sent to the target. When you perform an **Update Diagram**, changes to MATLAB workspace variables are sent.

If you select **Batch download**, the **Download** button is enabled. Until you click **Download**, changes made to block parameters are stored locally. When you click **Download**, the changes are sent in a single transmission.

When parameter changes are awaiting batch download, the External Mode Control Panel displays the message **Parameter changes pending...** to the right of the **Download** button. This message remains visible until the Simulink engine receives notification that the new parameters have been installed in the parameter vector of the target system.

The next figure shows the External Mode Control Panel with the **Batch download** option activated and parameter changes pending.



Configure Host Monitoring of Target Program Signal Data

- “Role of Trigger in Signal Data Uploading” on page 26-24
- “Configure Signal Data Uploading” on page 26-24
- “Default Trigger Options” on page 26-25
- “Select Signals to Upload” on page 26-26
- “Configure Trigger Options” on page 26-26
- “Select Trigger Signal” on page 26-28
- “Set Trigger Conditions” on page 26-29
- “Modify Signal and Triggering Options While Connected” on page 26-30

Role of Trigger in Signal Data Uploading

In external mode, uploading target program signal data to the host depends on a *trigger*. The trigger is a set of conditions that must be met for data uploading to begin. The trigger can be armed or not armed.

- When the trigger is armed, the software checks for the trigger conditions that allow data uploading to begin.
- If the trigger is not armed, the software does not check for the trigger conditions and data uploading cannot begin.
- The trigger can be armed automatically, when the host connects to the target, or manually, by clicking the **Arm Trigger** button on the External Mode Control Panel.

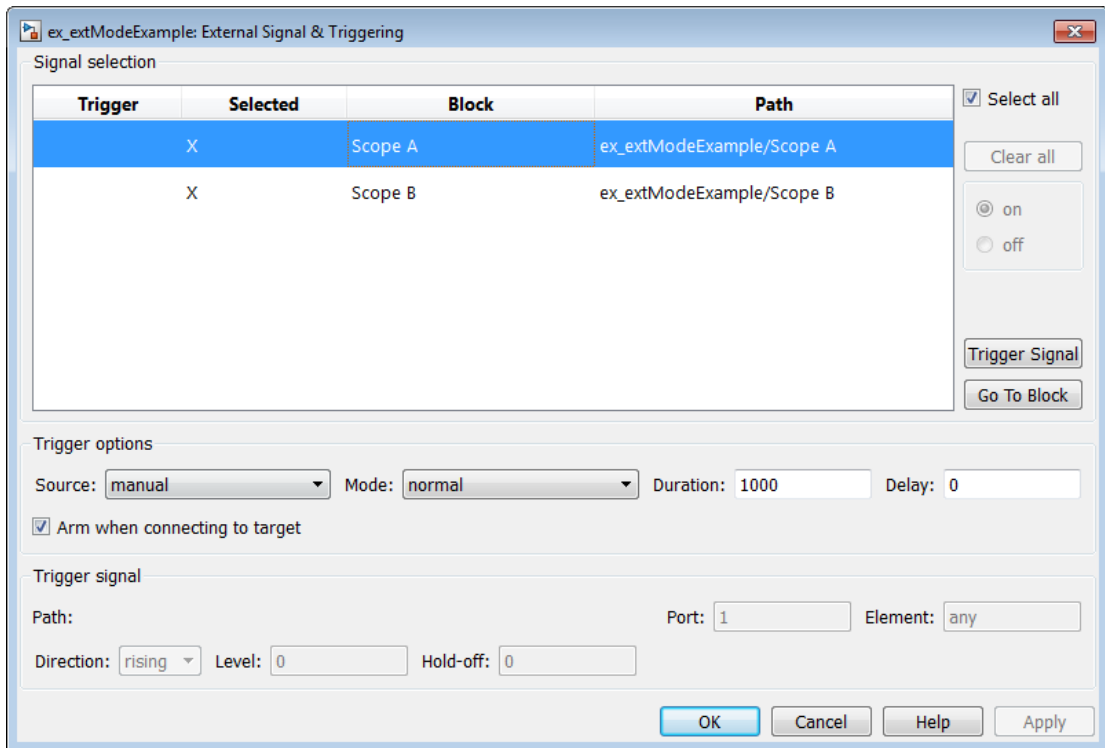
When the trigger is armed and the trigger conditions are met, the trigger fires and data uploading begins.

When data has been collected for a defined duration, the trigger event completes and data uploading stops. The trigger can then rearm, or remain unarmed until you click the **Arm Trigger** button.

To select the target program signals to upload and configure how uploads are triggered, see “Configure Signal Data Uploading” on page 26-24.

Configure Signal Data Uploading

Clicking the **Signal & Triggering** button of the External Mode Control Panel opens the External Signal & Triggering dialog box.



The External Signal & Triggering dialog box displays a list of blocks and subsystems in your model that support external mode signal uploading. For information on which types of blocks are external mode compatible, see “External Mode Compatible Blocks and Subsystems” on page 26-34.

In the External Signal & Triggering dialog box, you can select the signals that are collected from the target system and viewed in external mode. You can also select a trigger signal, which triggers uploading of data based on meeting certain signal conditions, and define the triggering conditions.

Default Trigger Options

The preceding figure shows the default settings of the External Signal & Triggering dialog box. The default operation of the External Signal & Triggering dialog box simplifies monitoring the target program. If you use the default settings, you do not need to preconfigure signals and triggers. You start the target program and connect the

Simulink engine to it. External mode compatible blocks are selected and the trigger is armed. Signal uploading begins immediately upon connection to the target program.

The default configuration of trigger options is:

- **Select all:** on
- **Source:** manual
- **Mode:** normal
- **Duration:** 1000
- **Delay:** 0
- **Arm when connecting to target:** on

Select Signals to Upload

External mode compatible blocks in your model appear in the **Signal selection** list of the External Signal & Triggering dialog box. You use this list to select signals that you want to view. In the **Selected** column, an X appears for each selected block.

The **Select all** check box selects all signals. By default, **Select all** is selected.

If **Select all** is cleared, you can select or clear individual signals using the **on** and **off** options. To select a signal, click its list entry and select the **on** option. To clear a signal, click its list entry and select the **off** option.

The **Clear all** button clears all signals.

Configure Trigger Options

As described in “Role of Trigger in Signal Data Uploading” on page 26-24, signal data uploading depends on a trigger. The trigger defines conditions that must be met for uploading to begin. Also, the trigger must be armed for data uploading to begin. When the trigger is armed and trigger conditions are met, the trigger fires and uploading begins. When data has been collected for a defined duration, the trigger event completes and data uploading stops.

To control when and how signal data is collected (uploaded) from the target system, configure the following **Trigger options** in the External Signal & Triggering dialog box.

- **Source:** manual or signal. Controls whether a button or a signal triggers data uploading.

Selecting **manual** directs external mode to use the **Arm Trigger** button on the External Mode Control Panel as the trigger to start uploading data. When you click **Arm Trigger**, data uploading begins.

Selecting **signal** directs external mode to use a trigger signal as the trigger to start uploading data. When the trigger signal satisfies trigger conditions (that is, the signal crosses the trigger level in the specified direction), a *trigger event* occurs. (Specify trigger conditions in the **Trigger signal** section.) If the trigger is *armed*, external mode monitors for the occurrence of a trigger event. When a trigger event occurs, data uploading begins.

- **Mode:** **normal** or **one-shot**. Controls whether the trigger rearms after a trigger event completes.

In **normal** mode, external mode automatically rearms the trigger after each trigger event. The next data upload begins when the trigger fires.

In **one-shot** mode, external mode collects only one buffer of data each time you arm the trigger.

For more information on the **Mode** setting, see “Configure Host Archiving of Target Program Signal Data” on page 26-31.

- **Duration:** Specifies the number of base rate steps for which external mode uploads data after a trigger event (default is 1000). For example, if **Duration** is set to 1000, and the base (fastest) rate of the model is one second:
 - For a signal sampled at the base rate, one second (1.0 Hz), external mode collects 1000 contiguous samples during a trigger event.
 - For a signal sampled at two seconds (0.5 Hz), external mode collects 500 samples during a trigger event.
- **Delay:** Specifies a delay to be applied to data collection. The delay represents the amount of time that elapses between a trigger event and the start of data collection. The delay is expressed in base rate steps. It can be positive or negative (default is 0). A negative delay corresponds to pretriggering. When the delay is negative, data from the time preceding the trigger event is collected and uploaded.
- **Arm when connecting to target:** Selected or cleared. Whether a button or a signal triggers data uploading (as defined by **Source**), the trigger must be armed to allow data uploading to begin.

If you select this option, connecting to the target arms the trigger.

- If the trigger **Source** is `manual`, data uploading begins immediately.
- If the trigger **Source** is `signal`, monitoring of the trigger signal begins immediately. Data uploading begins when the trigger signal satisfies trigger conditions (as defined in the **Trigger signal** section).

If you clear **Arm when connecting to target**, manually arm the trigger by clicking the **Arm Trigger** button on the External Mode Control Panel.

When simulating in external mode, each rate in the model creates a buffer on the target. Each entry in the buffer is big enough to hold all of the data required of every signal in that rate for one time step (time plus data plus external mode indices identifying the signal). The number of entries in the circular buffer is determined by the external mode trigger **Duration** parameter (`ExtModeTrigDuration`). The memory allocated on the target for buffering signals is proportional to the **Duration** and the number of signals uploading. The **Duration** also provides an indication of the number of base rate steps with log data after a trigger event in external mode.

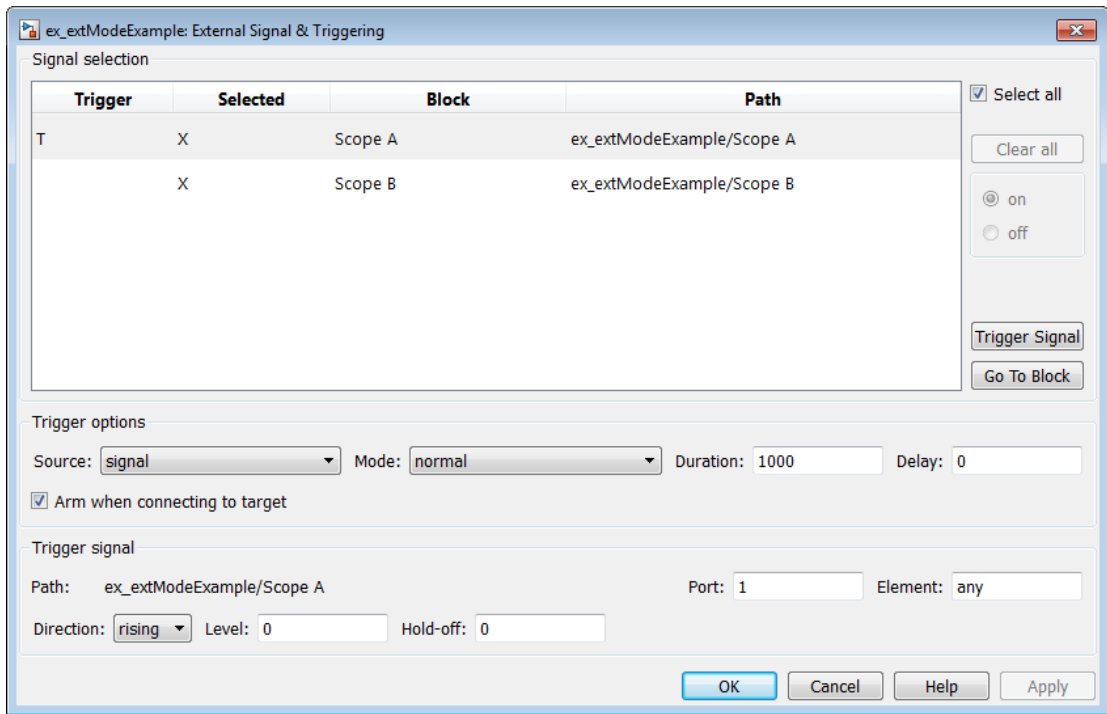
The **Duration** value specifies the number of contiguous points of data to be collected in each buffer of data. You should enter a **Duration** value equal to the number of continuous sample points that you need to collect rather than relying on a series of buffers to be continuous. If you enter a value less than the total number of sample points, you may lose sample points during the time spent transferring values from the data buffer to the MATLAB workspace. The Simulink software maintains point continuity only within one buffer. Between buffers, because of transfer time, some samples may be omitted.

The **Duration** value is related to the **Limit data points to last** value in the **Scope parameters** dialog box. The smaller of either value limits the number of sample points saved to the MATLAB workspace. You should not select the **Limit data points to last** check box; instead, use the **Duration** value to set the number of sample points saved.

Select Trigger Signal

You can designate one signal as a trigger signal. To select a trigger signal, from the **Source** menu in the **Trigger options** section, select `signal`. This action enables the parameters in the **Trigger signal** section. Then, select a signal in the **Signal selection** list, and click the **Trigger Signal** button.

When you select a signal to be a trigger, a T appears in the **Trigger** column of the **Signal selection** list. In the next figure, the **Scope A** signal is the trigger. **Scope B** is also selected for viewing, as indicated by the X in the **Selected** column.



After selecting the trigger signal, you can use the **Trigger signal** section to define the trigger conditions and set the trigger signal **Port** and **Element** parameters.

Set Trigger Conditions

Use the **Trigger signal** section of the External Signal & Triggering dialog box to set trigger conditions and attributes.

Note The **Trigger signal** parameters are enabled only when the trigger parameter **Source** is set to **signal** in the **Trigger options** section.

By default, any element of the first input port of a specified trigger block can cause the trigger to fire (that is, Port 1, any element). You can modify this behavior by adjusting the **Port** and **Element** values in the **Trigger signal** section. The **Port** field accepts a number or the keyword **last**. The **Element** field accepts a number or the keywords **any** or **last**.

In the **Trigger signal** section, you also define the conditions under which a trigger event occurs.

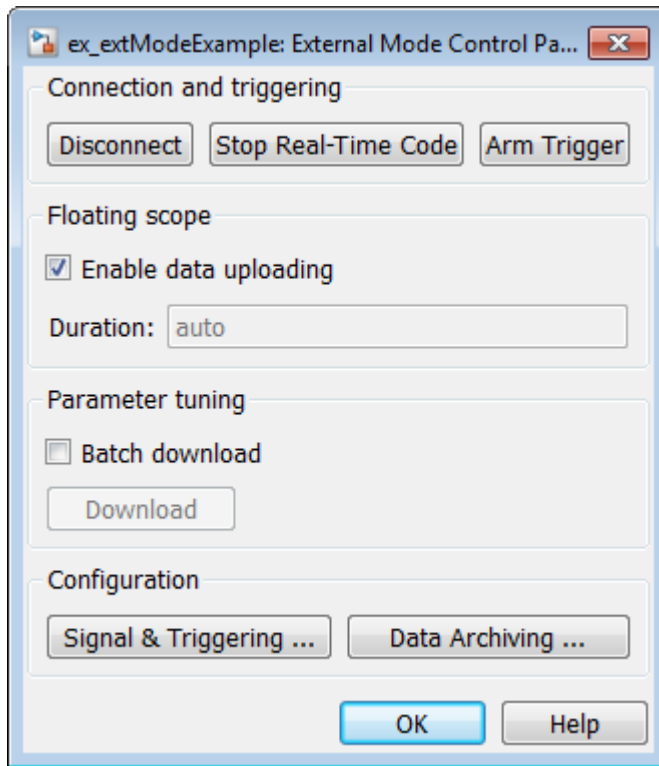
- **Direction:** `rising`, `falling`, or `either`. The direction in which the signal must be traveling when it crosses the threshold value. The default is `rising`.
- **Level:** A value indicating the threshold the signal must cross in a designated direction to fire the trigger. By default, the level is 0.
- **Hold-off:** Applies only to `normal` mode. Expressed in base rate steps, **Hold-off** is the time between the termination of one trigger event and the rearming of the trigger.

Modify Signal and Triggering Options While Connected

After you configure signal data uploading, and connect Simulink to a running target executable, you can modify signal and triggering options without disconnecting from the target.

If the trigger is armed (for example, if the trigger option **Arm when connecting to the target** is selected, which is the default), the External Signal & Triggering dialog box cannot be modified. To modify signal and triggering options:

- 1 Open the External Mode Control Panel.
- 2 Click **Cancel Trigger**. Triggering and display of uploaded data stops.
- 3 Open the External Signal & Triggering dialog box and modify signal and trigger options as required. For example, in the **Signal selection** section, you can enable or disable a scope, and in the **Trigger options** section, change the trigger **Mode**, for example, from `normal` to `one-shot`.
- 4 Click **Arm Trigger**. Triggering and display of uploaded data resumes, with your modifications.



Configure Host Archiving of Target Program Signal Data

In external mode, you can use the Simulink Scope and To Workspace blocks to archive data to disk.

To understand how the archiving features work, consider the handling of data when archiving is not enabled. There are two cases, one-shot mode and normal mode.

- In one-shot mode, after a trigger event occurs, each selected block writes its data to the workspace, as it would at the end of a simulation. If another one-shot is triggered, the existing workspace data is overwritten.
- In normal mode, external mode automatically rearms the trigger after each trigger event. Consequently, you can think of normal mode as a series of one-shots. Each one-shot in this series, except for the last, is referred to as an *intermediate result*. Because

the trigger can fire at any time, writing intermediate results to the workspace can result in unpredictable overwriting of the workspace variables. For this reason, the default behavior is to write only the results from the final one-shot to the workspace. The intermediate results are discarded. If you know that enough time exists between triggers for inspection of the intermediate results, you can override the default behavior by selecting the **Write intermediate results to workspace** option. This option does not protect the workspace data from being overwritten by subsequent triggers.

If you use a Simulink Scope block to archive data to disk, open the Scope parameters dialog box and select the option **Log data to workspace**. This is required for the following reasons:

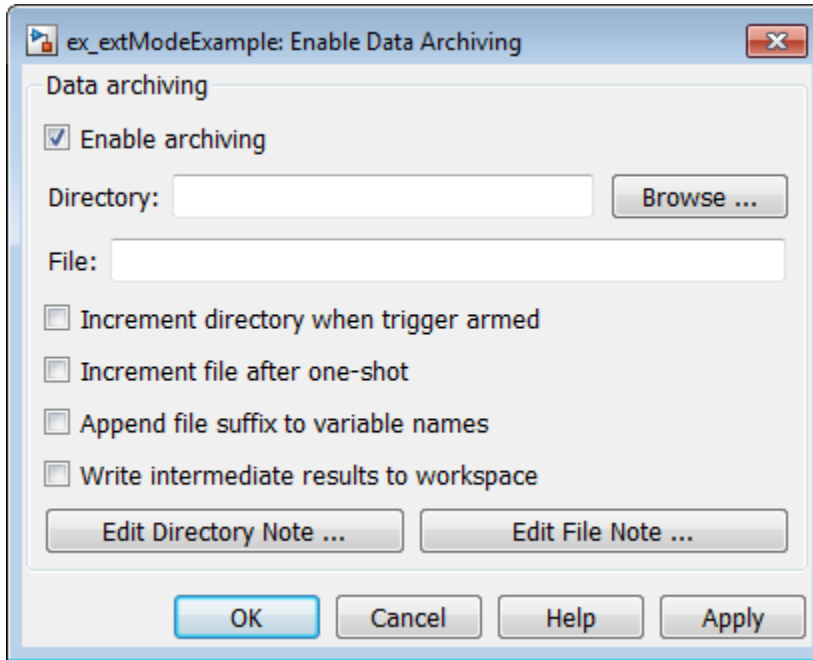
- The data is first transferred from the scope data buffer to the MATLAB workspace, before being written to a MAT-file.
- The **Variable name** entered in the Scope parameters dialog box is the same as the one in the MATLAB workspace and the MAT-file. Enabling the data to be saved enables a variable named with the **Variable name** parameter to be saved to a MAT-file.

Note: If you do not select the Scope block option **Log data to workspace**, the MAT-files for data logging are created, but they are empty.

The Enable Data Archiving dialog box supports the following capabilities:

- Folder notes
- File notes
- Automated data archiving

On the External Mode Control Panel, click the **Data Archiving** button to open the Enable Data Archiving dialog box. If your model is connected to the target environment, disconnect it while you configure data archiving. Select **Enable archiving** to enable the other controls in the dialog box.



The operations supported by the Enable Data Archiving dialog box are as follows.

Folder Notes

Use the **Edit Directory Note** button to add annotations for a collection of related data files in a folder. In the Enable Data Archiving dialog box, click **Edit Directory Note** to open the MATLAB editor. Place comments that you want saved to a file in the specified folder in this window. By default, the comments are saved to the folder last written to by data archiving.

File Notes

Use the **Edit File Note** button to add annotations for an individual data file. In the Enable Data Archiving dialog box, click **Edit File Note** to open a file finder window that is, by default, set to the last file to which you have written. Selecting a MAT-file opens an edit window. In this window, add or edit comments that you want saved with your individual MAT-file.

Automated Data Archiving

Use the **Enable archiving** option and the controls it enables to configure automatic writing of logging results, optionally including intermediate results, to disk. The dialog box provides the following related controls:

- **Directory:** Specifies the folder in which data is saved. If you select **Increment directory when trigger armed**, external mode appends a suffix.
- **File:** Specifies the name of the file in which data is saved. If you select **Increment file after one-shot**, external mode appends a suffix.
- **Increment directory when trigger armed:** Each time that you click the **Arm Trigger** button, external mode uses a different folder for writing log files. The folders are named incrementally, for example, `dirname1`, `dirname2`, and so on.
- **Increment file after one-shot:** New data buffers are saved in incremental files: `filename1`, `filename2`, and so on. File incrementing happens automatically in normal mode.
- **Append file suffix to variable names:** Whenever external mode increments file names, each file contains variables with identical names. Selecting **Append file suffix to variable name** results in each file containing unique variable names. For example, external mode saves a variable named `xdata` in incremental files (`file_1`, `file_2`, and so on) as `xdata_1`, `xdata_2`, and so on. This approach supports loading the MAT-files into the workspace and comparing variables at the MATLAB command prompt. Without the unique names, each instance of `xdata` would overwrite the previous one in the MATLAB workspace.
- **Write intermediate results to workspace:** If you want the Simulink Coder software to write intermediate results to the workspace, select this option.

External Mode Compatible Blocks and Subsystems

- “Compatible Blocks” on page 26-34
- “Signal Viewing Subsystems” on page 26-35
- “Supported Blocks for Data Archiving” on page 26-37

Compatible Blocks

In external mode, you can use the following types of blocks to receive and view signals uploaded from the target program:

- Floating Scope and Scope blocks

- Spectrum Analyzer, Time Scope, and Vector Scope blocks from the DSP System Toolbox product
- Display blocks
- To Workspace blocks
- User-written S-Function blocks

An external mode method is built into the S-function API. This method allows user-written blocks to support external mode. See `matlabroot/simulink/include/simstruc.h`.

- XY Graph blocks

In addition to these types of blocks, you can designate certain subsystems as Signal Viewing Subsystems and use them to receive and view signals uploaded from the target program. See “Signal Viewing Subsystems” on page 26-35 for more information.

You select external mode compatible blocks and subsystems, and arm the trigger, by using the External Signal & Triggering dialog box. By default, such blocks in a model are selected, and a manual trigger is set to be armed when connected to the target program.

Signal Viewing Subsystems

A Signal Viewing Subsystem is an atomic subsystem that encapsulates processing and viewing of signals received from the target system. A Signal Viewing Subsystem runs only on the host, and does not generate code in the target system. Signal Viewing Subsystems run in Normal, Accelerator, Rapid Accelerator, and External simulation modes.

Note: Signal Viewing Subsystems are inactive if placed inside a SIL or PIL component, such as a top model in SIL or PIL mode, a Model block in SIL or PIL mode, or a SIL or PIL block. However, a SIL or PIL component can feed a Signal Viewing Subsystem running in a supported mode.

Signal Viewing Subsystems are useful in situations where you want to process or condition signals before viewing or logging them, but you do not want to perform these tasks on the target system. By using a Signal Viewing Subsystem, you can generate smaller and more efficient code on the target system.

Like other external mode compatible blocks, Signal Viewing Subsystems are displayed in the External Signal & Triggering dialog box.

To declare a subsystem to be a Signal Viewing Subsystem,

- 1 Select the **Treat as atomic unit** option in the Block Parameters dialog box.

For more information on atomic subsystems, see “Code Generation of Subsystems” on page 6-2.

- 2 Use the following `set_param` command to turn the `SimViewingDevice` property on,

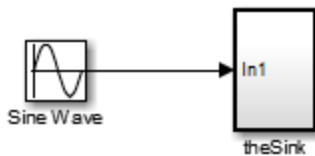
```
set_param('blockname', 'SimViewingDevice', 'on')
```

where `'blockname'` is the name of the subsystem.

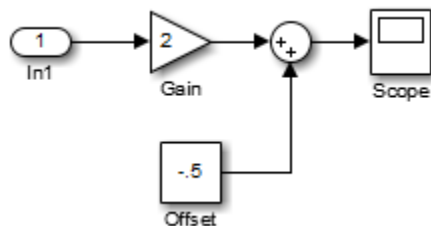
- 3 Make sure the subsystem meets the following requirements:

- It must be a pure Sink block. That is, it must not contain Outport blocks or Data Store blocks. It can contain Goto blocks only if the corresponding From blocks are contained within the subsystem boundaries.
- It must not have continuous states.

The following model, `sink_examp`, contains an atomic subsystem, `theSink`.



The subsystem `theSink`, shown in the next figure, applies a gain and an offset to its input signal and displays it on a Scope block.



If `theSink` is declared as a Signal Viewing Subsystem, the generated target program includes only the code for the Sine Wave block. If `theSink` is selected and armed in

the External Signal & Triggering dialog box, the target program uploads the sine wave signal to theSink during simulation. You can then modify the parameters of the blocks within theSink and observe the uploaded signal.

If theSink were not declared as a Signal Viewing Subsystem, its Gain, Constant, and Sum blocks would run as subsystem code on the target system. The Sine Wave signal would be uploaded to the Simulink engine after being processed by these blocks, and viewed on sink_examp/theSink/Scope2. Processing demands on the target system would be increased by the additional signal processing, and by the downloading of changes in block parameters from the host.

Supported Blocks for Data Archiving

In external mode, you can use the following types of blocks to archive data to disk:

- Scope blocks
- To Workspace blocks

You configure data archiving by using the Enable Data Archiving dialog box, as described in “Configure Host Archiving of Target Program Signal Data” on page 26-31.

External Mode Communication

- “About External Mode Communication” on page 26-37
- “Download Mechanism” on page 26-37
- “Inlined and Tunable Parameters” on page 26-39

About External Mode Communication

Depending on the setting of the **Default parameter behavior** option when the target program is generated, there are differences in the way parameter updates are handled. “Download Mechanism” on page 26-37 describes the operation of external mode communication with **Default parameter behavior** set to Tunable. “Inlined and Tunable Parameters” on page 26-39 describes the operation of external mode with **Default parameter behavior** set to Inlined.

Download Mechanism

In external mode, the Simulink engine does not simulate the system represented by the block diagram. By default, when external mode is enabled, the Simulink engine

downloads parameters to the target system. After the initial download, the engine remains in a waiting mode until you change parameters in the block diagram or until the engine receives data from the target.

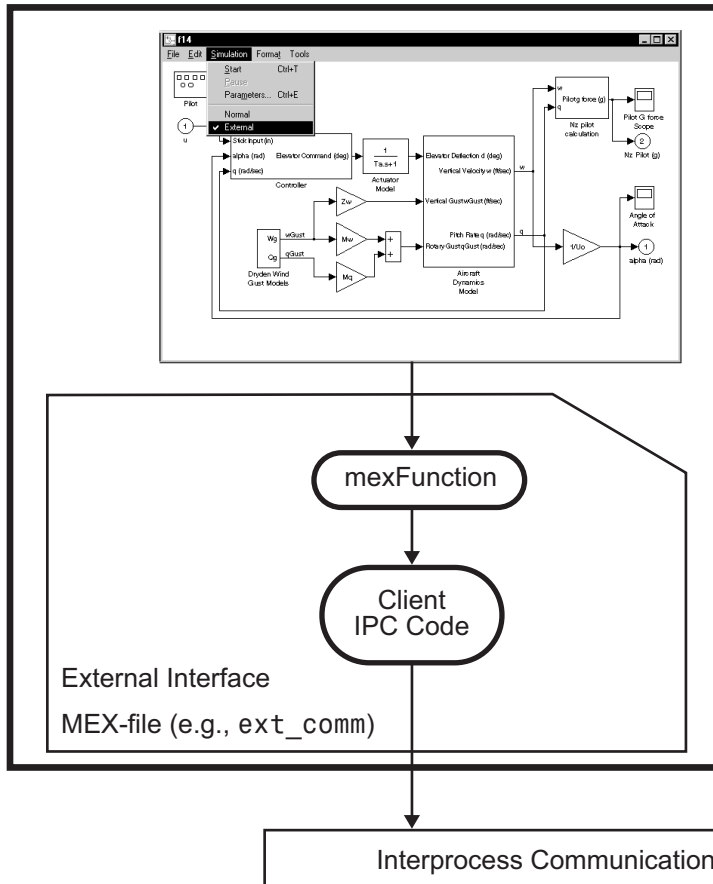
When you change a parameter in the block diagram, the Simulink engine calls the external interface MEX-file, passing new parameter values (along with other information) as arguments. The external interface MEX-file contains code that implements one side of the interprocess communication (IPC) channel. This channel connects the Simulink process (where the MEX-file executes) to the process that is executing the external program. The MEX-file transfers the new parameter values by using this channel to the external program.

The other side of the communication channel is implemented within the external program. This side writes the new parameter values into the target's parameter structure (*model_P*).

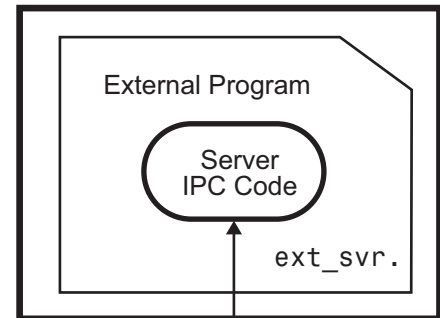
The Simulink side initiates the parameter download operation by sending a message containing parameter information to the external program. In the terminology of client/server computing, the Simulink side is the client and the external program is the server. The two processes can be remote, or they can be local. Where the client and server are remote, a protocol such as TCP/IP is used to transfer data. Where the client and server are local, a serial connection or shared memory can be used to transfer data.

The next figure shows this relationship. The Simulink engine calls the external interface MEX-file whenever you change parameters in the block diagram. The MEX-file then downloads the parameters to the external program by using the communication channel.

Simulink Process



External Program Process



External Mode Architecture

Inlined and Tunable Parameters

By default, parameters (except those listed in “External Mode Limitations” on page 26-55) in an external mode program are tunable; that is, you can change them by using the download mechanism described in this section.

If you set **Default parameter behavior** to **Inlined** (on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box), the Simulink Coder

code generator embeds the numerical values of model parameters (constants), instead of symbolic parameter names, in the generated code. Inlining parameters generates smaller and more efficient code. However, inlined parameters, because they effectively become constants, are not tunable.

The Simulink Coder software lets you improve overall efficiency by inlining most parameters, while at the same time retaining the flexibility of run-time tuning for selected parameters that are important to your application. When you inline parameters, you can use `Simulink.Parameter` objects to remove individual parameters from inlining and declare them to be tunable. In addition, you can use these objects to control how parameters are represented in the generated code.

For more information on tunable parameters, see “Block Parameter Representation in the Generated Code” on page 14-43.

Automatic Parameter Uploading on Host/Target Connection

Each time the Simulink engine connects to a target program that was generated with **Default parameter behavior** set to `Inlined`, the target program uploads the current value of its tunable parameters to the host. These values are assigned to the corresponding MATLAB workspace variables. This procedure synchronizes the host and target with respect to parameter values.

Workspace variables required by the model must be initialized at the time of host/target connection. Otherwise the uploading cannot proceed and an error results. Once the connection is made, these variables are updated to reflect the current parameter values on the target system.

Automatic parameter uploading takes place only if the target program was generated with **Default parameter behavior** set to `Inlined`. “Download Mechanism” on page 26-37 describes the operation of external mode communication with **Default parameter behavior** set to `Tunable`.

Choose Communication Protocol for Client and Server

- “Introduction” on page 26-41
- “Using the TCP/IP Implementation” on page 26-41
- “Using the Serial Implementation” on page 26-44
- “Run the External Program” on page 26-46

- “Implement an External Mode Protocol Layer” on page 26-48

Introduction

The Simulink Coder product provides code to implement both the client and server side of external mode communication using either TCP/IP or serial protocols. You can use the socket-based external mode implementation provided by the Simulink Coder product with the generated code, provided that your target system supports TCP/IP. If not, use or customize the serial transport layer option provided.

A low-level *transport layer* handles physical transmission of messages. Both the Simulink engine and the model code are independent of this layer. Both the transport layer and code directly interfacing to the transport layer are isolated in separate modules that format, transmit, and receive messages and data packets.

See “Target Interfacing” on page 26-16 for information on selecting a transport layer.

Using the TCP/IP Implementation

You can use TCP/IP-based client/server implementation of external mode with real-time programs on The Open Group UNIX or PC systems. For help in customizing external mode transport layers, see “Create a Transport Layer for External Communication” on page 36-8.

To use Simulink external mode over TCP/IP:

- Make sure that the external interface MEX-file for your target's TCP/IP transport is specified.

Targets provided by MathWorks specify the name of the external interface MEX-file in `matlabroot/toolbox/simulink/simulink/extmode_transports.m`. The name of the interface appears as uneditable text in the **External mode configuration** section of the **Interface** pane of the Configuration Parameters dialog box. The TCP/IP default is `ext_comm`.

To specify a TCP/IP transport for a custom target, you must add an entry of the following form to an `sl_customization.m` file on the MATLAB path:

```
function sl_customization(cm)
    cm.ExtModeTransports.add('stf.tlc', 'transport', 'mexfile', 'Level1');
%end function
```

where

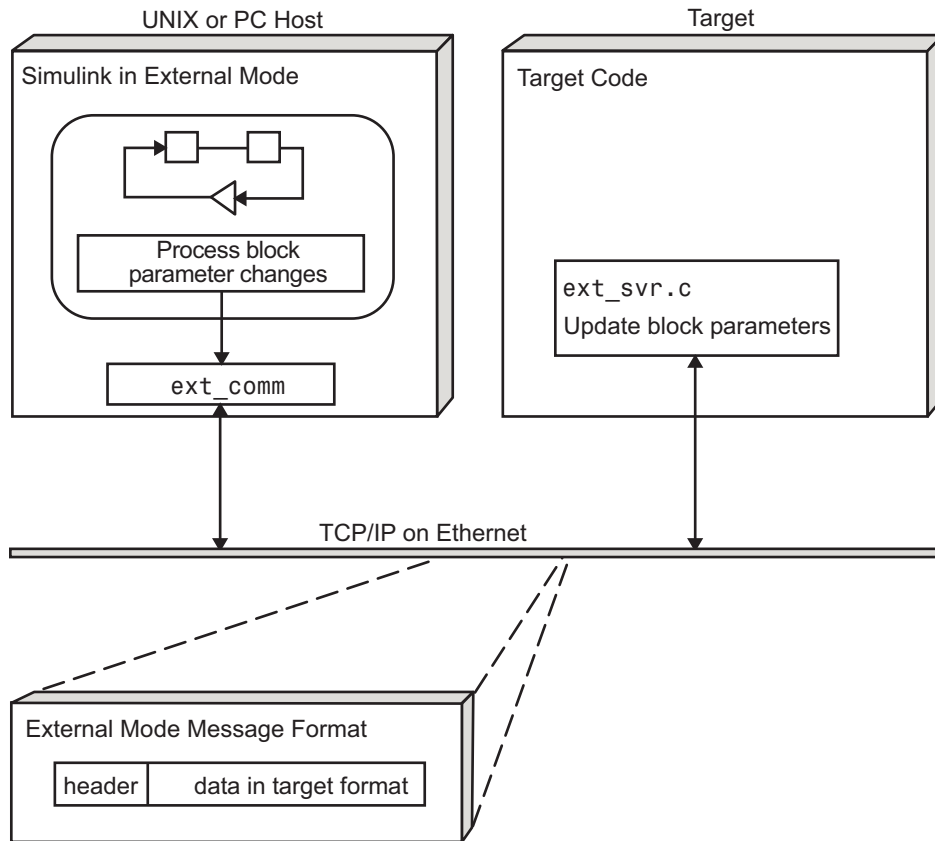
- *stf.tlc* is the name of the system target file for which you are registering the transport (for example, 'mytarget.tlc')
- *transport* is the transport name to display in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box (for example, 'tcpip')
- *mexfile* is the name of the transport's associated external interface MEX-file (for example, 'ext_comm')

You can specify multiple targets and/or transports with additional `cm.ExtModeTransports.add` lines, for example:

```
function sl_customization(cm)
    cm.ExtModeTransports.add('mytarget.tlc', 'tcpip', 'ext_comm', 'Level1');
    cm.ExtModeTransports.add('mytarget.tlc', 'serial', ...
                             'ext_serial_win32_comm', 'Level1');
%end function
```

- Be sure that the template makefile is configured to link the source files for the TCP/IP server code and that it defines the compiler flags when building the generated code.
- Build the external program.
- Run the external program.
- Set the Simulink model to external mode and connect to the target.

The next figure shows the structure of the TCP/IP-based implementation.



TCP/IP-Based Client/Server Implementation for External Mode

MEX-File Optional Arguments for TCP/IP Transport

In the External Target Interface dialog box, you can specify optional arguments that are passed to the external mode interface MEX-file for communicating with executing targets.

- Target network name: the network name of the computer running the external program. By default, this is the computer on which the Simulink product is running, for example, 'myComputer'. You can also use the IP address, for example, '148.27.151.12'.

- **Verbosity level:** controls the level of detail of the information displayed during the data transfer. The value is either 0 or 1 and has the following meaning:
 - 0 — No information
 - 1 — Detailed information
- **TCP/IP server port number:** The default value is 17725. You can change the port number to a value between 256 and 65535 to avoid a port conflict.

The arguments are positional and must be specified in the following order:

```
<TargetNetworkName> <VerbosityLevel> <ServerPortNumber>
```

For example, if you want to specify the verbosity level (the second argument), then you must also specify the target network name (the first argument). Arguments can be delimited by white space or commas. For example:

```
'148.27.151.12' 1 30000
```

You can specify command-line options to the external program when you launch it. See “Run the External Program” on page 26-46 for more information.

Using the Serial Implementation

Controlling host/target communication on a serial channel is similar to controlling host/target communication on a TCP/IP channel.

To use Simulink external mode over a serial channel, you must

- Make sure that the external interface MEX-file for your target's serial transport is specified.

Targets provided by MathWorks specify the name of the external interface MEX-file in `matlabroot/toolbox/simulink/simulink/extmode_transports.m`.

The name of the interface appears as uneditable text in the **External mode configuration** section of the **Interface** pane of the Configuration Parameters dialog box. The serial default is `serial`.

To specify a serial transport for a custom target, you must add an entry of the following form to an `sl_customization.m` file on the MATLAB path:

```
function sl_customization(cm)
    cm.ExtModeTransports.add('stf.tlc', 'transport', 'mexfile', 'Level1');
%end function
```

where

- *stf.tlc* is the name of the system target file for which you are registering the transport (for example, 'mytarget.tlc')
- *transport* is the transport name to display in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box (for example, 'serial')
- *mexfile* is the name of the transport's associated external interface MEX-file (for example, 'ext_serial_win32_comm')

You can specify multiple targets and/or transports with additional `cm.ExtModeTransports.add` lines, for example:

```
function sl_customization(cm)
    cm.ExtModeTransports.add('mytarget.tlc', 'tcpip', 'ext_comm', 'Level1');
    cm.ExtModeTransports.add('mytarget.tlc', 'serial', ...
        'ext_serial_win32_comm', 'Level1');
%end function
```

- Be sure that the template makefile is configured to link the source files for the serial server code and that it defines the compiler flags when building the generated code.
- Build the external program.
- Run the external program.
- Set the Simulink model to external mode and connect to the target.

MEX-File Optional Arguments for Serial Transport

In the **MEX-file arguments** field of the **Interface** pane of the Configuration Parameters dialog box, you can specify arguments that are passed to the external mode interface MEX-file for communicating with the executing targets. For serial transport, the optional arguments to `ext_serial_win32_comm` are as follows:

- Verbosity level: This argument controls the level of detail of the information displayed during data transfer. The value of this argument is
 - 0 (no information), or
 - 1 (detailed information)
- Serial port ID: The port ID of the host, specified as an integer or character vector. For example, specify the port ID of a USB to serial converter as '/dev/ttyusb0'.

Note: Simulink Coder prefixes integer port IDs with '\\.\COM on Windows and by /dev/ttyS on Unix.

When you start the target program using a serial connection, you must specify the port ID to use to connect it to the host. Do this by including the `-port` command-line option. For example,

```
mytarget.exe -port 2 -w
```

- Baud rate: Specify an integer value. The default value is 57600.

The MEX-file options arguments are positional and must be specified in the following order:

```
<VerbosityLevel> <SerialPortID> <BaudRate>
```

For example, if you want to specify the serial port ID (the second argument), then you must also specify the verbosity level (the first argument). Arguments can be delimited by white space or commas. For example:

```
1 '/dev/ttyusb0' 57600
```

You can specify command-line options to the external program when you launch it. The following section provides more information on using command-line arguments.

Run the External Program

The external program must be running before you can use the Simulink product in external mode.

If the target program is executing on the same machine as the host and communication is through a loopback serial cable, the target's port ID must differ from that of the host (as specified in the **MEX-file arguments** edit field).

To run the external program, you type a command of the form

```
model -opt1 ... -optN
```

where *model* is the name of the external program and *-opt1 ... -optN* are options. (See “Command-Line Options for the External Program” on page 26-47.) In the examples in this section, the name of the external program is assumed to be `ext_example`.

Running the External Program Under the Windows Environment

In the Windows environment, you can run the external programs in either of the following ways:

- Open a Command Prompt window. At the command prompt, type the name of the target executable, followed by possible options, such as:

```
ext_example -tf inf -w
```

- Alternatively, you can launch the target executable from the MATLAB Command Window. The command must be preceded by an exclamation point (!) and followed by an ampersand (&), as in the following example:

```
!ext_example -tf inf -w &
```

The ampersand (&) causes the operating system to spawn another process to run the target executable. If you do not include the ampersand, the program still runs, but you cannot enter commands at the MATLAB command prompt or manually terminate the executable.

Running the External Program Under the UNIX Environment

In the UNIX environment, you can run the external programs in either of the following ways:

- Open an Xterm window. At the command prompt, type the name of the target executable, followed by possible options, such as:

```
./ext_example -tf inf -w
```

- Alternatively, you can launch the target executable from the MATLAB Command Window. You must run it in the background so that you can still access the Simulink environment. The command must be preceded by an exclamation point (!), dot slash (./ indicating the current directory), and followed by an ampersand (&), as in the following example:

```
!./ext_example -tf inf -w &
```

The ampersand (&) causes the operating system to spawn another process to run the target executable.

Command-Line Options for the External Program

External mode target executables generated by the Simulink Coder code generator support the following command-line options:

- `-tf n`

The `-tf` option overrides the stop time set in the Simulink model. The argument `n` specifies the number of seconds the program will run. The value `inf` directs the model to run indefinitely. In this case, the model code will run until the target program receives a stop message from the Simulink engine.

The following example sets the stop time to 10 seconds.

```
ext_example -tf 10
```

When integer-only ERT targets are built and executed in external mode, the stop time parameter (`-tf`) is interpreted by the target as the number of base rate ticks rather than the number of seconds to execute.

- `-w`

Instructs the target program to enter a wait state until it receives a message from the host. At this point, the target is running, but not executing the model code. The start message is sent when you select **Start Real-Time Code** from the **Simulation** menu or click the **Start Real-Time Code** button in the External Mode Control Panel.

Use the `-w` option if you want to view data from time step 0 of the target program execution, or if you want to modify parameters before the target program begins execution of model code.

- `-port n`

Specifies the TCP/IP port number or the serial port ID, `n`, for the target program. The port number of the target program must match that of the host for TCP/IP transport. The port number depends on the type of transport.

- For TCP/IP transport: Port number is an integer between 256 and 65535, with the default value being 17725.
- For serial transport: Port ID is an integer or a character vector. For example, specify the port ID of a USB to serial converter as `'/dev/ttyusb0'`

- `-baud r`

Specified as an integer, this option is only available for serial transport.

Implement an External Mode Protocol Layer

If you want to implement your own transport layer for external mode communication, you must modify certain code modules provided by the Simulink Coder product and

create a new external interface MEX-file. This advanced topic is addressed in “Create a Transport Layer for External Communication” on page 36-8.

Use External Mode Programmatically

You can run external-mode applications from the MATLAB command line or programmatically in scripts. Use the `get_param` and `set_param` commands to retrieve and set the values of model simulation command-line parameters, such as `SimulationMode` and `SimulationCommand`, and external mode command-line parameters, such as `ExtModeCommand` and `ExtModeTrigType`.

The following model simulation commands assume that a Simulink model is open and that you have loaded a target program to which the model will connect using external mode.

- 1 Change the Simulink model to external mode:

```
set_param(gcs, 'SimulationMode', 'external')
```
- 2 Connect the open model to the loaded target program:

```
set_param(gcs, 'SimulationCommand', 'connect')
```
- 3 Start running the target program:

```
set_param(gcs, 'SimulationCommand', 'start')
```
- 4 Stop running the target program:

```
set_param(gcs, 'SimulationCommand', 'stop')
```
- 5 Disconnect the target program from the model:

```
set_param(gcs, 'SimulationCommand', 'disconnect')
```

To tune a workspace parameter, change its value at the command prompt. If the workspace parameter is a `Simulink.Parameter` object, assign the new value to the `Value` property.

```
myVariable = 5.23;  
myParamObj.Value = 5.23;
```

To download the workspace parameter in external mode, you update the model diagram. The following model simulation command initiates a model update:

```
set_param(gcs, 'SimulationCommand', 'update')
```

To trigger or cancel data uploading to scopes, use the `ExtModeCommand` values `armFloating` and `cancelFloating`, or `armWired` and `cancelWired`. For example, to trigger and then cancel data uploading to wired (nonfloating) scopes:

```
set_param(gcs, 'ExtModeCommand', 'armWired')
set_param(gcs, 'ExtModeCommand', 'cancelWired')
```

The next table lists external mode command-line parameters that you can use in `get_param` and `set_param` commands. The table provides brief descriptions, valid values (bold type highlights defaults), and a mapping to External Mode dialog box equivalents.

Note: For external mode parameters that are equivalent to **Interface** pane options in the Configuration Parameters dialog box, see “Model Configuration Parameters: Code Generation Interface”.

External Mode Command-Line Parameters

Parameter and Values	Dialog Box Equivalent	Description
<code>ExtModeAddSuffixToVar</code> off , on	Enable Data Archiving: Append file suffix to variable names check box	Increment variable names for each incremented filename.
<code>ExtModeArchiveDirName</code> <i>character vector</i>	Enable Data Archiving: Directory text field	Save data in specified folder.
<code>ExtModeArchiveFileName</code> <i>character vector</i>	Enable Data Archiving: File text field	Save data in specified file.
<code>ExtModeArchiveMode</code> <i>character vector</i> - off , on	Enable Data Archiving: Enable archiving check box	Activate automated data archiving features.
<code>ExtModeArmWhenConnect</code> off, on	External Signal & Triggering: Arm when connecting to target check box	Arm the trigger as soon as the Simulink Coder software connects to the target.
<code>ExtModeAutoIncOneShot</code> off , on	Enable Data Archiving: Increment file after one-shot check box	Save new data buffers in incremental files.
<code>ExtModeAutoUpdateStatusClock</code> (Microsoft Windows platforms only)	Not available	Continuously upload and display target time on the model window status bar.

Parameter and Values	Dialog Box Equivalent	Description
off, on		
ExtModeBatchMode off, on	External Mode Control Panel: Batch download check box	Enable or disable downloading of parameters in batch mode.
ExtModeChangesPending off, on	Not available	When ExtModeBatchMode is enabled, indicates whether parameters remain in the queue of parameters to be downloaded to the target.
ExtModeCommand <i>character vector</i> - armFloating, armWired, cancelFloating, cancelWired	<ul style="list-style-type: none"> armFloating and cancelFloating are equivalent to selecting and clearing External Mode Control Panel check box Floating scope > Enable data uploading armWired and cancelWired are equivalent to External Mode Control Panel buttons Arm Trigger and Cancel Trigger 	Issue an external mode command to the target program.
ExtModeConnected off, on	External Mode Control Panel: Connect/Disconnect button	Indicate the state of the connection with the target program.
ExtModeEnableFloating off, on	External Mode Control Panel: Enable data uploading check box	Enable or disable the arming and canceling of triggers when a connection is established with floating scopes.
ExtModeIncDirWhenArm off, on	Enable Data Archiving: Increment directory when trigger armed check box	Write log files to incremental folders each time the trigger is armed.

Parameter and Values	Dialog Box Equivalent	Description
ExtModeLogAll off, on	External Signal & Triggering: Select all check box	Upload available signals from the target to the host.
ExtModeParamChangesPending off, on	Not available	When the Simulink Coder software is connected to the target and ExtModeBatchMode is enabled, indicates whether parameters remain in the queue of parameters to be downloaded to the target. More efficient than ExtModeChangesPending, because it checks for a connection to the target.
ExtModeSkipDownloadWhenConnect off, on	Not available	Connect to the target program without downloading parameters.
ExtModeTrigDelay integer (0)	External Signal & Triggering: Delay text field	Specify the amount of time (expressed in base rate steps) that elapses between a trigger occurrence and the start of data collection.
ExtModeTrigDirection character vector - rising , falling, either	External Signal & Triggering: Direction menu	Specify the direction in which the signal must be traveling when it crosses the threshold value.
ExtModeTrigDuration integer (1000)	External Signal & Triggering: Duration text field	Specify the number of base rate steps for which external mode is to log data after a trigger event.
ExtModeTrigDurationFloating character vector - integer (auto)	External Mode Control Panel: Duration text field	Specify the duration for floating scopes. If auto is specified, the value of ExtModeTrigDuration is used.

Parameter and Values	Dialog Box Equivalent	Description
ExtModeTrigElement <i>character vector - integer, any, last</i>	External Signal & Triggering: Element text field	Specify the elements of the input port of the specified trigger block that can cause the trigger to fire.
ExtModeTrigHoldOff <i>integer (0)</i>	External Signal & Triggering: Hold-off text field	Specify the base rate steps between when a trigger event terminates and the trigger is rearmed.
ExtModeTrigLevel <i>integer (0)</i>	External Signal & Triggering: Level text field	Specify the threshold value the trigger signal must cross to fire the trigger.
ExtModeTrigMode <i>character vector - normal, oneshot</i>	External Signal & Triggering: Mode menu	Specify whether the trigger is to rearm automatically after each trigger event or whether only one buffer of data is to be collected each time the trigger is armed.
ExtModeTrigPort <i>character vector - integer (1), last</i>	External Signal & Triggering: Port text field	Specify the input port of the specified trigger block for which elements can cause the trigger to fire.
ExtModeTrigType <i>character vector - manual, signal</i>	External Signal & Triggering: Source menu	Specify whether to start logging data when the trigger is armed or when a specified trigger signal satisfies trigger conditions.
ExtModeUploadStatus <i>character vector - inactive, armed, uploading</i>	Not available	Return the status of the external mode upload mechanism — inactive, armed, or uploading.
ExtModeWriteAllDataToWs off , on	Enable Data Archiving: Write intermediate results to workspace check box	Write intermediate results to the workspace.

Animate Stateflow Charts in External Mode

If you have a Stateflow license, you can animate a chart in external mode. In external mode, you can animate states in a chart, and view test point signals in a floating scope or signal viewer.

- “Animate States During Simulation in External Mode” on page 26-54
- “View Test Point Data in Floating Scopes and Signal Viewers” on page 26-55

Animate States During Simulation in External Mode

To animate states in a chart in external mode:

- 1 Load the chart you want to animate to the target machine.
- 2 Open the Model Configuration Parameters dialog box.
- 3 In the left Select pane, select **Code Generation > Interface**.
- 4 In the right pane, **Data exchange interface** section, select **External mode** and click **OK**.
- 5 In the Simulink Editor, select **Code > External Mode Control Panel**.
- 6 In the External Mode Control Panel dialog box, click **Signal & Triggering**.
- 7 In the External Signal & Triggering dialog box, set these parameters:

In:	Select:
Signal selection pane	Chart you want to animate
Trigger pane	Arm when connecting to target check box
Trigger pane	normal from drop-down menu in Mode field

- 8 Build the model to generate an executable file.
- 9 Start the target in the background by typing this command at the MATLAB prompt:

```
!model_name.exe -w &
```

For example, if the name of your model is `my_control_sys`, enter this command:

```
!my_control_sys.exe -w &
```

Note: `-w` allows the target code to wait for the Simulink model connection.

- 10 In the Model Editor, select **Simulation > Mode > External**, and then select **Simulation > Connect to Target**.
- 11 Start simulation.

The chart highlights states as they execute.

View Test Point Data in Floating Scopes and Signal Viewers

When you simulate a chart in external mode, you can designate chart data of local scope to be test points and view the test point data in floating scopes and signal viewers.

To view test point data during simulation in external mode:

- 1 Open the Model Explorer and for each data you want to view, follow these steps:
 - a In the middle **Contents** pane, select the state or local data of interest.
 - b In the right **Dialog** pane, select the **Logging** tab and select **Test point** check box.
- 2 From a floating scope or signal viewer, click the signal selection button:



The Signal Selector dialog box opens.

- 3 In the Signal Selector **Model hierarchy** pane, select the chart.
- 4 In the Signal Selector **List contents** menu, select **Testpointed/Logged signals only** and then select the signals you want to view.
- 5 Simulate the model in external mode as described in “Animate States During Simulation in External Mode” on page 26-54.

The scope or viewer displays the values of the test point signals as the simulation runs.

For more information, see “Behavior of Scopes and Viewers with Rapid Accelerator Mode” in the Simulink documentation.

External Mode Limitations

- “Limitation on Changing Parameters” on page 26-56
- “Limitation on Mixing 32-bit and 64-bit Architectures” on page 26-57

- “Limitation on Uploading Data” on page 26-57
- “Limitation on Uploading Variable-Size Signals” on page 26-57
- “Limitation on Signal Value Display in Simulation” on page 26-57
- “Limitation on Tunable Structure Parameters” on page 26-57
- “Limitation on Archiving Data” on page 26-57
- “Limitation on Scopes in Referenced Models” on page 26-58
- “Limitation on Simulation Start Time” on page 26-58
- “Limitation on File-Scoped Data” on page 26-58

Limitation on Changing Parameters

In general, you cannot change a parameter if doing so results in a change in the structure of the model. For example, you cannot change

- The number of states, inputs, or outputs of a block
- The sample time or the number of sample times
- The integration algorithm for continuous systems
- The name of the model or of a block
- The parameters to the Fcn block

If you make these changes to the block diagram, then you must rebuild the program with newly generated code.

However, you can change parameters in transfer function and state space representation blocks in specific ways:

- The parameters (numerator and denominator polynomials) for the Transfer Fcn (continuous and discrete) and Discrete Filter blocks can be changed (as long as the number of states does not change).
- Zero entries in the State-Space and Zero Pole (both continuous and discrete) blocks in the user-specified or computed parameters (that is, the A, B, C, and D matrices obtained by a zero-pole to state-space transformation) cannot be changed once external simulation is started.
- In the State-Space block, if you specify the matrices in the controllable canonical realization, then all changes to the A, B, C, D matrices that preserve this realization and the dimensions of the matrices are allowed.

If the Simulink block diagram does not match the external program, the Simulink engine displays an error informing you that the checksums do not match (that is, the model has

changed since you generated code). This means that you must rebuild the program from the new block diagram (or reload another one) to use external mode.

If the external program is not running, the Simulink engine displays an error informing you that it cannot connect to the external program.

Limitation on Mixing 32-bit and 64-bit Architectures

When you use external mode, the machine running the Simulink product and the machine running the target executable must have matching bit architectures, either 32-bit or 64-bit. This is because the Simulink Coder software varies a model's checksum based on whether it is configured for a 32-bit or 64-bit platform.

If you attempt to connect from a 32-bit machine to a 64-bit machine or vice versa, the external mode connection fails.

Limitation on Uploading Data

External mode does not support uploading data values for fixed-point or enumerated types into workspace parameters.

Limitation on Uploading Variable-Size Signals

External mode does not support uploading variable-size signals for the following targets:

- Simulink Real-Time
- Texas Instruments™ C2000™

Limitation on Signal Value Display in Simulation

External mode does not support graphical display of signal values in models (described in “Displaying Signal Values in Model Diagrams”). For example, you cannot use the **Data Display in Simulation** menu selections **Show Value Labels When Hovering**, **Toggle Value Labels When Clicked**, and **Show Value Label of Selected Port**.

Limitation on Tunable Structure Parameters

External mode does not support uploading or downloading tunable structure parameters.

Limitation on Archiving Data

External mode supports the Scope and To Workspace blocks for archiving data to disk. However, external mode does not support scopes other than the Scope block for archiving

data. For example, you cannot use Floating Scope blocks or Signal and Scope Manager viewer objects to archive data in external mode.

Limitation on Scopes in Referenced Models

In a model hierarchy, if the top model simulates in external mode and a referenced model simulates in Normal or Accelerator mode, scopes in the referenced model do not display.

However, if the top model is changed to simulate in Normal mode, the behavior of scopes in the referenced models differs between Normal and Accelerator mode. Scopes in a referenced model simulating in Normal mode display, while scopes in a referenced model simulating in Accelerator mode do not display.

Limitation on Simulation Start Time

External mode does not support nonzero simulation start times. In the Configuration Parameters dialog box, **Solver** pane, leave **Start time** set to the default value of **0.0**.

Limitation on File-Scoped Data

External mode does not support file-scoped data, for example, data items to which you apply the built-in custom storage class `FileScope`. File-scoped data are not externally accessible.

Related Examples

- “Use External Mode with the ERT Target”

Logging in Simulink Coder

Log Program Execution Results

Multiple techniques are available by which a program generated by the Simulink Coder software can save data to a MAT-file for analysis. A generated executable can save system states, outputs, and simulation time at each model execution time step. The data is written to a MAT-file, named (by default) *model.mat*, where *model* is the name of your model. See “Log Data for Analysis” on page 27-2 for a data logging tutorial.

Note: Data logging is available only for system target files that have access to a file system. In addition, only the RSim target executables are capable of accessing MATLAB workspace data.

For MAT-file logging limitations, see the configuration parameter “MAT-file logging”.

In this section...
“Log Data for Analysis” on page 27-2
“Configure State, Time, and Output Logging” on page 27-9
“Log Data with Scope and To Workspace Blocks” on page 27-11
“Log Data with To File Blocks” on page 27-11
“Data Logging Differences Between Single- and Multitasking” on page 27-12

Log Data for Analysis

- “Set Up and Configure Model” on page 27-2
- “Data Logging During Simulation” on page 27-4
- “Data Logging from Generated Code” on page 27-7

Set Up and Configure Model

This example shows how data generated by a copy of the model `slexAircraftExample` is logged to the file `myAircraftExample.mat`. Refer to “Build a Generic Real-Time Program” on page 25-23 for instructions on setting up a copy of `slexAircraftExample` `asmAircraftExample` in a working folder if you have not done so already.

Note: When you configure the code generator to produce code that includes support for data logging during execution, the code generator can include text for block names in the block paths included in the log file. If the text includes characters that are unrepresented in the character set encoding for the model, the code generator replaces the characters with XML escape sequences. For example, the code generator replaces the Japanese full-width Katakana letter `ヲ` with the escape sequence `ア`; . For more information, see “Internationalization and Code Generation” on page 18-2.

To configure data logging, open the Configuration Parameters dialog box and select the **Data Import/Export** pane. The process is the same as configuring a Simulink model to save output to the MATLAB workspace. For each workspace return variable you define and enable, the Simulink Coder software defines a parallel MAT-file variable. For example, if you save simulation time to the variable `tout`, your generated program logs the same data to a variable named `rt_tout`. You can change the prefix `rt_` to a suffix (`_rt`), or eliminate it entirely. You do this by setting **Configuration Parameters > All Parameters > MAT-file variable name modifier**.

Simulink lets you log signal data from anywhere in a model. In the Simulink Editor, select the signals that you want to log and then in the **Simulation Data Inspector** button dropdown, select **Log Selected Signals to Workspace**. However, the Simulink Coder software does not use this method of signal logging in generated code. To log signals in generated code, you must either use the **Data Import/Export** options described below or include To File or To Workspace blocks in your model.

Note: If you enable MAT-file and signal logging (through the **Data Import/Export** pane) and select signals for logging (through the Simulink Editor), you see the following warning when you build the model:

```
Warning: MAT-file logging does not support signal logging.  
When your model code executes, the signal logging variable 'rt_logout' will  
not be saved to the MAT-file.
```

To avoid this warning, clear the **Data Import/Export > Signal logging** check box.

In this example, you modify the `myAircraftExample` model so that the generated program saves the simulation time and system outputs to the file `myAircraftExample.mat`. Then you load the data into the base workspace and plot simulation time against one of the outputs. The `myAircraftExample` model should be configured as described in “Build a Generic Real-Time Program” on page 25-23.

Data Logging During Simulation

To use the data logging feature:

- 1 Open the `myAircraftExample` model if it is not already open.
- 2 Open the Configuration Parameters dialog box by selecting **Simulation > Model Configuration Parameters** from the model window.
- 3 Select the **Data Import/Export** pane. The **Data Import/Export** pane lets you specify which output data is to be saved to the workspace and what variable names to use for it.
- 4 Set **Format** to **Structure with time**. When you select this format, Simulink saves the model states and outputs in structures that have their names specified in the **Save to workspace or file** area. By default, the structures are `xout` for states and `yout` for output. The structure used to save output has two top-level fields: `time` and `signals`. The `time` field contains a vector of simulation times and `signals` contains an array of substructures, each of which corresponds to a model output port.
- 5 Select the **Output** option. This tells Simulink to save output signal data during simulation as a variable named `yout`. Selecting **Output** enables the code generator to create code that logs the root Output block (`alpha`, `rad`) to a MAT-file.
- 6 Set **Decimation** to 1.
- 7 If other options are enabled, clear them. The figure below shows how the dialog box should appear.

Load from workspace

Input:

Initial state:

Save to workspace or file

Time:

States: Format:

Output:

Final states: Save complete SimState in final state

Signal logging:

Data stores:

Log Dataset data to file:

Single simulation output: Logging intervals:

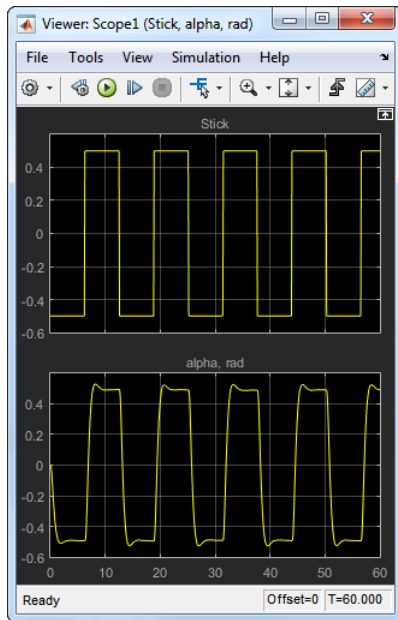
Simulation Data Inspector

Record logged workspace data in Simulation Data Inspector

Write streamed signals to workspace

► Additional parameters

- 8 Click **Apply** and **OK** to register your changes and close the dialog box.
- 9 Save the model.
- 10 In the model window, double-click the scope symbol next to the Aircraft Dynamics Model block, then run the model by choosing **Simulation** > **Run** in the model window. The resulting scope display is shown below.



- 11** Verify that the simulation time and outputs have been saved to the base workspace in MAT-files. At the MATLAB prompt, type:

```
whos yout
```

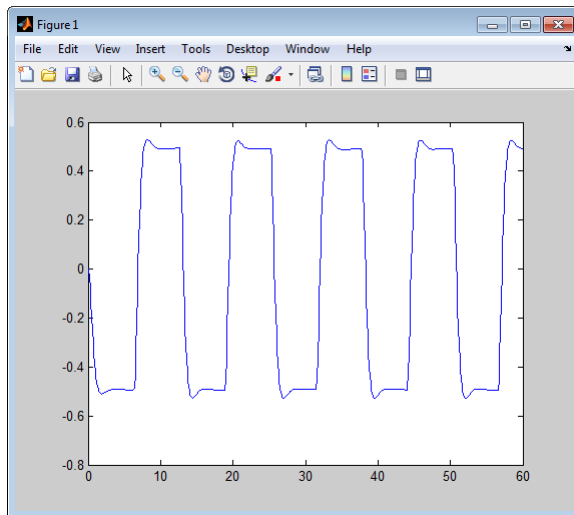
Simulink displays:

Name	Size	Bytes	Class	Attributes
yout	1x1	10756	struct	

- 12** Verify that `alpha, rad` was logged by plotting simulation time versus that variable. In the Command Window, type:

```
plot(yout.time,yout.signals.values)
```

The resulting plot is shown below.



Data Logging from Generated Code

In the second part of this example, you build and run a Simulink Coder executable of the `myAircraftExample` model that outputs a MAT-file containing the simulation time and output you previously examined. Even though you have already generated code for the `myAircraftExample` model, you must now regenerate that code because you have changed the model by enabling data logging. The steps below explain this procedure.

To avoid overwriting workspace data with data from simulation runs, the code generator modifies identifiers for variables logged by Simulink. You can control these modifications.

- 1 Set **Configuration Parameters** > **All Parameters** > **MAT-file variable name modifier** to `_rt`. This adds the suffix `_rt` to each variable that you selected to be logged in the first part of this example.
- 2 Click **Apply** and **OK** to register your changes and close the dialog box.
- 3 Save the model.
- 4 Build an executable.
- 5 When the build concludes, run the executable with the command:

```
!myAircraftExample
```
- 6 The program now produces two message lines, indicating that the MAT-file has been written.

```
** starting the model **  
** created myAircraftExample.mat **
```

- 7 Load the MAT-file data created by the executable and look at the workspace variables from simulation and the generated program by typing:

```
load myAircraftExample.mat  
whos yout*
```

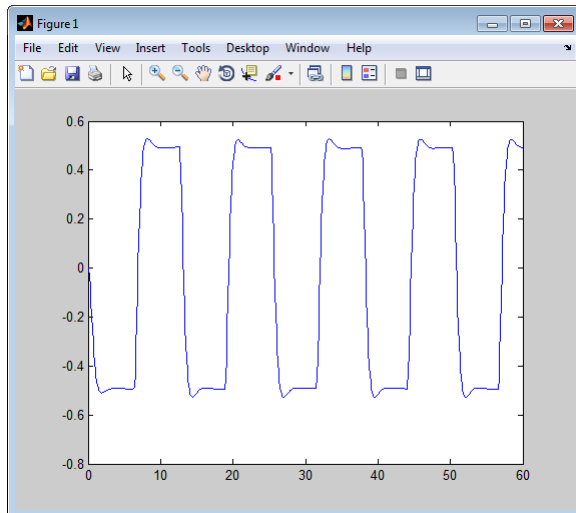
Simulink displays:

Name	Size	Bytes	Class	Attributes
yout	1x1	10756	struct	
yout_rt	1x1	10756	struct	

Note the size and bytes of the structures resulting from the simulation run and generated code are the same.

- 8 Plot the generated code output by entering the following command in the Command Window:

```
plot(yout_rt.time,yout_rt.signals.values)
```



The plot should be identical to the plot that you produced in the previous part of this example.

Tip For UNIX platforms, run the executable in the Command Window with the syntax `!./executable_name`. If preferred, run the executable from an OS shell with the syntax `./executable_name`. For more information, see “Run External Commands, Scripts, and Programs”.

Configure State, Time, and Output Logging

The **Data Import/Export** pane enables a generated program to save system states, outputs, and simulation time at each model execution time step. The data is written to a MAT-file, named (by default) `model.mat`.

Before using this data logging feature, you should learn how to configure a Simulink model to return output to the MATLAB workspace. This is discussed in “Export Simulation Data”.

For each workspace return variable that you define and enable, the code generator defines a MAT-file variable. For example, if your model saves simulation time to the workspace variable `tout`, your generated program logs the same data to a variable named (by default) `rt_tout`.

The code generated by the code generator logs the following data:

- Root Outport blocks

The default MAT-file variable name for system outputs is `rt_yout`.

The sort order of the `rt_yout` array is based on the port number of the Outport block, starting with 1.

- Continuous and discrete states in the model

The default MAT-file variable name for system states is `rt_xout`.

- Simulation time

The default MAT-file variable name for simulation time is `rt_tout`.

- “Override Default MAT-File Variable Names” on page 27-10

- “Override Default MAT-File Name or Buffer Size” on page 27-10

Override Default MAT-File Variable Names

By default, the code generation software prefixes the text `rt_` to the variable names for system outputs, states, and simulation time to form MAT-file variable names. To change this prefix for a model, select a prefix (`rt_`), a suffix (`_rt`), or no modifier (`none`) for **Configuration Parameters > All Parameters > MAT-file variable name modifier**. Other system target files might not support this parameter.

Override Default MAT-File Name or Buffer Size

You can specify compiler options to override the following MAT-file attributes in generated code:

MAT-File Attribute	Default	Compiler Option
Name	<code>model.mat</code>	<code>-DSAVEFILE=<i>filename</i></code>
Size of data logging buffer	1024 bytes	<code>-DDEFAULT_BUFFER_SIZE=<i>n</i></code>

Note: Valid option syntax can vary among compilers. For example, Microsoft Visual C++ compilers typically accept `/DSAVEFILE=filename` as well as `-DSAVEFILE=filename`.

For a template makefile (TMF) based target, append the compiler option to the **Make command** field on the **Code Generation** pane of the Configuration Parameters dialog box. For example:

Make command: `make_rtw OPTS="-DSAVEFILE=myCodeLog.mat"`

For a toolchain-based target such as GRT or ERT, add the compiler option to the **Build configuration** settings on the **Code Generation** pane of the Configuration Parameters dialog box. Set **Build configuration** to **Specify**, and add the compiler option to the **C Compiler** row of the **Tool/Options** table. For example:

Tool	Options
C Compiler	<code>\$(cflags) \$(CVARSFLAG) \$(CFLAGS_ADDITIONAL) -DSAVEFILE=myCodeLog.mat /Od /Oy-</code>

To add the compiler option to a custom toolchain, you can modify and reregister the custom toolchain using the procedures shown in the example “Adding a Custom

Toolchain”. For example, to add the compiler option to the MATLAB source file for the custom toolchain, you could define `myCompilerOpts` as follows:

```

optimsOffOpts = {'/c /Od'};
optimsOnOpts  = {'/c /O2'};
cCompilerOpts = '$(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL)';
cppCompilerOpts = '$(cflags) $(CVARSFLAG) $(CPPFLAGS_ADDITIONAL)';
myCompilerOpts = {' -DSAVEFILE=myCodeLog.mat '};
...

```

Then you can add `myCompilerOpts` to the flags for each configuration and compiler to which it applies, for example:

```

cfg = tc.getBuildConfiguration('Faster Builds');
cfg.setOption('C Compiler', horzcat(cCompilerOpts, myCompilerOpts, optimsOffOpts));

```

As shown in “Adding a Custom Toolchain”, after modifying the custom toolchain, you save the configuration to a MAT-file and refresh the target registry.

Log Data with Scope and To Workspace Blocks

The code generated by the code generator also logs data from these sources:

- Scope blocks that have the **Log data to workspace** parameter enabled
 - You must specify the variable name and data format in each Scope block's dialog box.
- To Workspace blocks in the model
 - You must specify the variable name and data format in each To Workspace block's dialog box.

The variables are written to `model.mat`, along with variables logged from the **Workspace I/O** pane.

Log Data with To File Blocks

You can also log data to a To File block. The generated program creates a separate MAT-file (distinct from `model.mat`) for each To File block in the model. The file contains the block time and input variable(s). You must specify the filename, variable names, decimation, and sample time in the To File block dialog box.

Note: Models referenced by Model blocks do not perform data logging in that context except for states, which you can include in the state logged for top models. Code

generated by the Simulink Coder software for referenced models does not perform data logging to MAT-files.

Data Logging Differences Between Single- and Multitasking

When logging data in single-tasking and multitasking systems, you will notice differences in the logging of

- Noncontinuous root Outport blocks
- Discrete states

In multitasking mode, the logging of states and outputs is done after the first task execution (and not at the end of the first time step). In single-tasking mode, the code generated by the build procedure logs states and outputs after the first time step.

See Data Logging in Single-Tasking and Multitasking Model Execution on page 15-12 for more details on the differences between single-tasking and multitasking data logging.

Note: The rapid simulation target (RSim) provides enhanced logging options. See “Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File” on page 22-2 for more information.

Data Interchange Using the C API in Simulink Coder

- “Exchange Data Between Generated and External Code Using C API” on page 28-2
- “Use C API to Access Model Signals and States” on page 28-24
- “Use C API to Access Model Parameters” on page 28-30

Exchange Data Between Generated and External Code Using C API

Some Simulink Coder applications must interact with signals, states, root-level inputs/outputs, or parameters in the generated code for a model. For example, calibration applications monitor and modify parameters. Signal monitoring or data logging applications interface with signal, state, and root-level input/output data. Using the Simulink Coder C API, you can build target applications that log signals, states, and root-level inputs/outputs, monitor signals, states, and root-level inputs/outputs, and tune parameters, while the generated code executes.

The C API minimizes its memory footprint by sharing information common to signals, states, root-level inputs/outputs, and parameters in smaller structures. Signal, state, root-level input/output, and parameter structures include an index into the structure map, allowing multiple signals, states, root-level inputs/outputs, or parameters to share data.

To get started with an example, see “Use C API to Access Model Signals and States” on page 28-24 or “Use C API to Access Model Parameters” on page 28-30.

In this section...

“Generated C API Files” on page 28-2

“Generate C API Files” on page 28-3

“Description of C API Files” on page 28-5

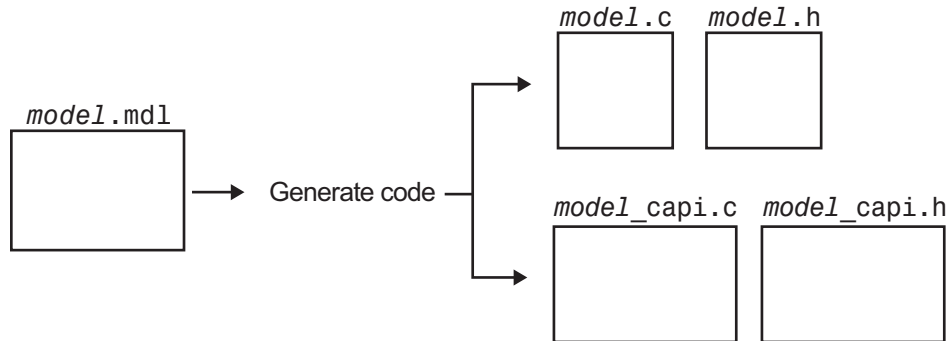
“Generate C API Data Definition File for Exchanging Data with a Target System” on page 28-20

“C API Limitations” on page 28-22

Generated C API Files

When you configure a model to use the C API, the Simulink Coder code generator generates two additional files, *model_capi.c* (or *.cpp*) and *model_capi.h*, where *model* is the name of the model. The code generator places the two C API files in the build folder, based on settings in the Configuration Parameters dialog box. The C API source code file contains information about global block output signals, states, root-level inputs/outputs, and global parameters defined in the generated code model source code. The C API header file is an interface header file between the model source code and

the generated C API. You can use the information in these C API files to create your application. Among the files generated are those shown in the next figure.



Generated Files with C API Selected

Note: When you configure the code generator to produce code that includes support for the C API interface and data logging, the code generator can include text for block names in the block paths logged to C API files *model_capi.c* (or *.cpp*) and *model_capi.h*. If the text includes characters that are unrepresented in the character set encoding for the model, the code generator replaces the characters with XML escape sequences. For example, the code generator replaces the Japanese full-width Katakana letter ア with the escape sequence `ア`. For more information, see “Internationalization and Code Generation” on page 18-2.

Generate C API Files

To generate C API files for your model:

- 1 Select the C API interface for your model. There are two ways to select the C API interface for your model, as described in the following sections.
 - “Select C API with Configuration Parameters Dialog” on page 28-4
 - “Select C API from the Command Line” on page 28-4
- 2 Generate code for your model.

After generating code, you can examine the files *model_capi.c* (or *.cpp*) and *model_capi.h* in the model build folder.

Select C API with Configuration Parameters Dialog

- 1 Open your model, and open the Configuration Parameters dialog box.
- 2 Go to the **Code Generation > Interface** pane and, in the **Data exchange interface** subgroup, select one or more C API options. Based on the options you select, support for accessing signals, parameters, states, and root-level I/O will appear in the C API generated code.
 - If you want to generate C API code for global block output signals, select **Generate C API for: signals**.
 - If you want to generate C API code for global block parameters, select **Generate C API for: parameters**.
 - If you want to generate C API code for discrete and continuous states, select **Generate C API for: states**.
 - If you want to generate C API code for root-level inputs and outputs, select **Generate C API for: root-level I/O**.

Select C API from the Command Line

From the MATLAB command line, you can use the `set_param` function to select or clear the C API check boxes on the **Interface** pane of the Configuration Parameters dialog box. At the MATLAB command line, enter one or more of the following commands, where `modelName` is the name of your model.

To select **Generate C API for: signals**, enter:

```
set_param('modelName', 'RTWCAPISignals', 'on')
```

To clear **Generate C API for: signals**, enter:

```
set_param('modelName', 'RTWCAPISignals', 'off')
```

To select **Generate C API for: parameters**, enter:

```
set_param('modelName', 'RTWCAPIParams', 'on')
```

To clear **Generate C API for: parameters**, enter:

```
set_param('modelName', 'RTWCAPIParams', 'off')
```

To select **Generate C API for: states**, enter:

```
set_param('modelName', 'RTWCAPISStates', 'on')
```

To clear **Generate C API for: states**, enter:

```
set_param('modelName', 'RTWCAPISStates', 'off')
```

To select **Generate C API for: root-level I/O**, enter:

```
set_param('modelName', 'RTWCAPIRootIO', 'on')
```

To clear **Generate C API for: root-level I/O**, enter:

```
set_param('modelName', 'RTWCAPIRootIO', 'off')
```

Description of C API Files

- “About C API Files” on page 28-5
- “Structure Arrays Generated in C API Files” on page 28-8
- “Generate Example C API Files” on page 28-9
- “C API Signals” on page 28-11
- “C API States” on page 28-14
- “C API Root-Level Inputs and Outputs” on page 28-15
- “C API Parameters” on page 28-16
- “Map C API Data Structures to rtModel” on page 28-18

About C API Files

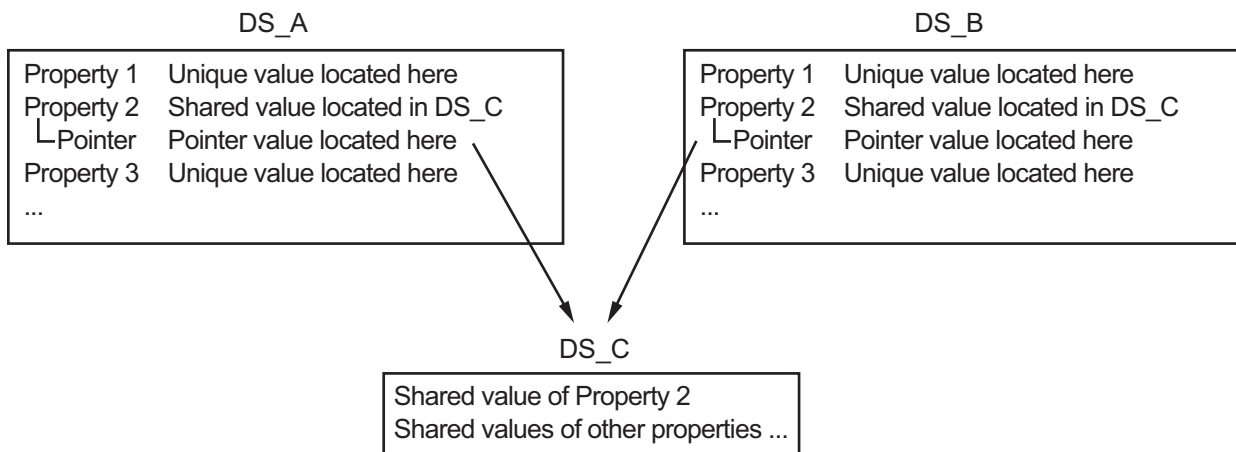
The `model_capi.c` (or `.cpp`) file provides external applications with a consistent interface to model data. Depending on your configuration settings, the data could be a signal, state, root-level input or output, or parameter. In this document, the term *data item* refers to either a signal, a state, a root-level input or output, or a parameter. The C API uses structures that provide an interface to the data item properties. The interface packages the properties of each data item in a data structure. If the model contains multiple data items, the interface generates an array of data structures. The members of a data structure map to data properties.

To interface with data items, an application requires the following properties for each data item:

- Name

- Block path
- Port number (for signals and root-level inputs/outputs only)
- Address
- Data type information: native data type, data size, complexity, and other attributes
- Dimensions information: number of rows, number of columns, and data orientation (scalar, vector, matrix, or n -dimensional)
- Fixed-point information: slope, bias, scale type, word length, exponent, and other attributes
- Sample-time information (for signals, states, and root-level inputs/outputs only): sample time, task identifier, frames

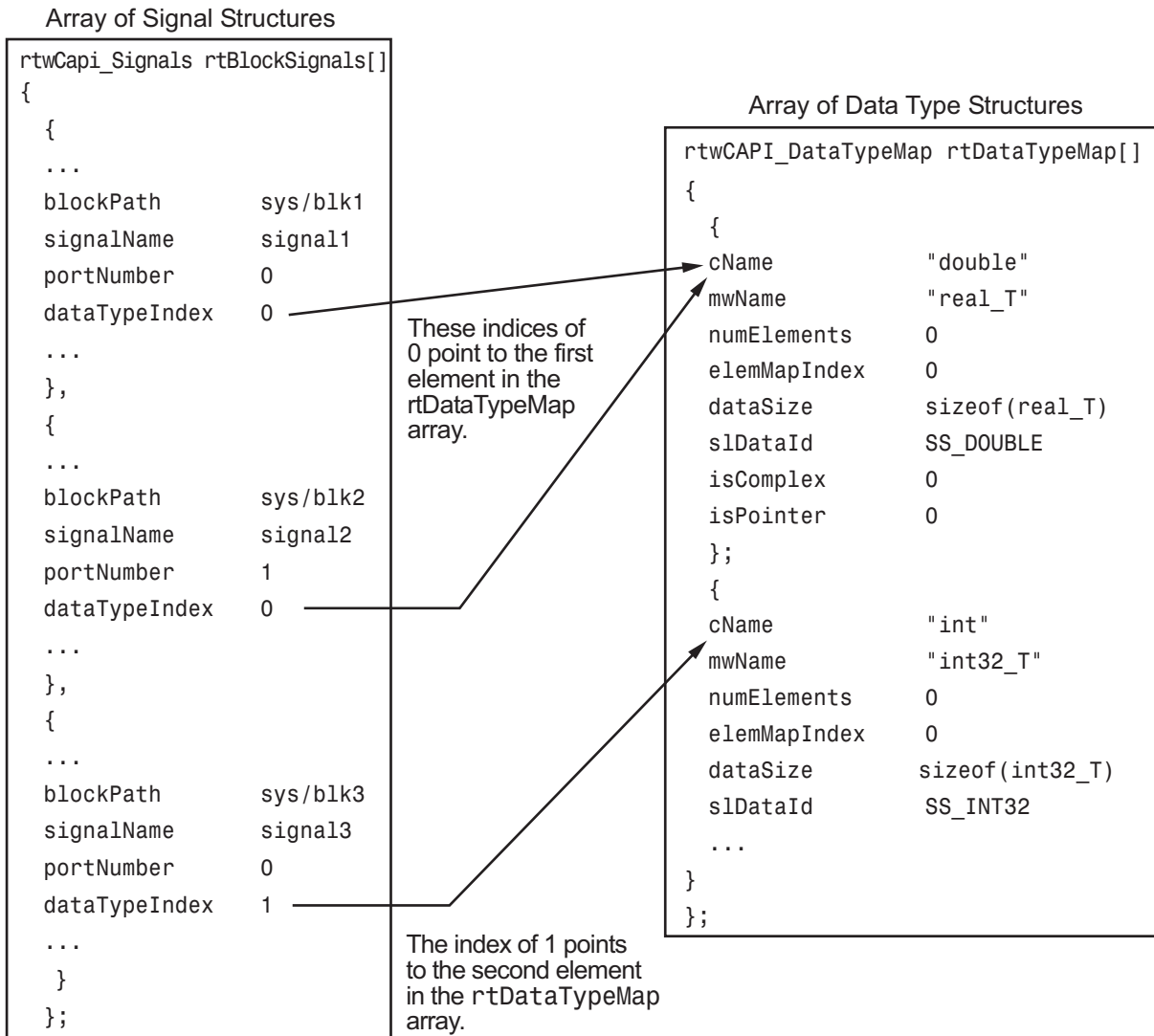
As illustrated in the next figure, the properties of data item A, for example, are located in data structure DS_A. The properties of data item B are located in data structure DS_B.



Some property *values* can be unique to each data item, and there are some property values that several data items can share in common. Name, for example, has a unique value for each data item. The interface places the unique property values directly in the structure for the data item. The name value of data item A is in DS_A, and the name value of data item B is in DS_B.

But data type could be a property whose value several data items have in common. The ability of some data items to share a property allows the C API to have a reuse feature. In this case, the interface places only an index value in DS_A and an index value in

DS_B. These indices point to a different data structure, DS_C, that contains the actual data type value. The next figure shows this scheme with more detail.



The figure shows three signals. **signal1** and **signal2** share the same data type, **double**. Instead of specifying this data type value in each signal data structure, the

interface provides only an index value, 0, in the structure. "double" is described by entry 0 in the `rtDataTypeMap` array, which is referenced by both signals. Additionally, property values can be shared between signals, states, root-level inputs/outputs, and parameters, so states, root-level inputs/outputs, and parameters also might reference the `double` entry in the `rtDataTypeMap` array. This reuse of information reduces the memory size of the generated interface.

Structure Arrays Generated in C API Files

As with data type, the interface maps other common properties (such as address, dimension, fixed-point scaling, and sample time) into separate structures and provides an index in the structure for the data item. For a complete list of structure definitions, refer to the file `matlabroot/rtw/c/src/rtw_capi.h`. This file also describes each member in a structure. The structure arrays generated in the `model_capi.c` (or `.cpp`) file are of structure types defined in the `rtw_capi.h` file. Here is a brief description of the structure arrays generated in `model_capi.c` (or `.cpp`):

- **rtBlockSignals** is an array of structures that contains information about global block output signals in the model. Each element in the array is of type `struct rtwC_API_Signals`. The members of this structure provide the signal name, block path, block port number, address, and indices to the data type, dimension, fixed-point, and sample-time structure arrays.
- **rtBlockParameters** is an array of structures that contains information about the tunable block parameters in the model by block name and parameter name. Each element in the array is of type `struct rtwC_API_BlockParameters`. The members of this structure provide the parameter name, block path, address, and indices to data type, dimension, and fixed-point structure arrays.
- **rtBlockStates** is an array of structures that contains information about discrete and continuous states in the model. Each element in the array is of type `struct rtwC_API_States`. The members of this structure provide the state name, block path, type (continuous or discrete), and indices to the address, data type, dimension, fixed-point, and sample-time structure arrays.
- **rtRootInputs** is an array of structures that contains information about root-level inputs in the model. Each element in the array is of type `struct rtwC_API_Signals`. The members of this structure provide the root-level input name, block path, block port number, address, and indices to the data type, dimension, fixed-point, and sample-time structure arrays.
- **rtRootOutputs** is an array of structures that contains information about root-level outputs in the model. Each element in the array is of type `struct`

`rtwC_API_Signals`. The members of this structure provide the root-level output name, block path, block port number, address, and indices to the data type, dimension, fixed-point, and sample-time structure arrays.

- **`rtwC_API_ModelParameters`** is an array of structures that contains information about workplace variables that one or more blocks or Stateflow charts in the model reference as block parameters. Each element in the array is of data type `rtwC_API_ModelParameters`. The members of this structure provide the variable name, address, and indices to data type, dimension, and fixed-point structure arrays.
- **`rtwC_API_DataAddrMap`** is an array of base addresses of signals, states, root-level inputs/outputs, and parameters that appear in the `rtwC_API_BlockSignals`, `rtwC_API_BlockParameters`, `rtwC_API_BlockStates`, and `rtwC_API_ModelParameters` arrays. Each element of the `rtwC_API_DataAddrMap` array is a pointer to `void (void*)`.
- **`rtwC_API_DataTypeMap`** is an array of structures that contains information about the various data types in the model. Each element of this array is of type `struct rtwC_API_DataTypeMap`. The members of this structure provide the data type name, size of the data type, and information on whether or not the data is complex.
- **`rtwC_API_DimensionMap`** is an array of structures that contains information about the various data dimensions in the model. Each element of this array is of type `struct rtwC_API_DimensionMap`. The members of this structure provide information on the number of dimensions in the data, the orientation of the data (whether it is scalar, vector, or a matrix), and the actual dimensions of the data.
- **`rtwC_API_FixPtMap`** is an array of structures that contains fixed-point information about the signals, states, root-level inputs/outputs, and parameters. Each element of this array is of type `struct rtwC_API_FixPtMap`. The members of this structure provide information about the data scaling, bias, exponent, and whether or not the fixed-point data is signed. If the model does not have fixed-point data (signal, state, root-level input/output, or parameter), the Simulink Coder software assigns NULL or zero values to the elements of the `rtwC_API_FixPtMap` array.
- **`rtwC_API_SampleTimeMap`** is an array of structures that contains sampling information about the global signals, states, and root-level inputs/outputs in the model. (This array does not contain information about parameters.) Each element of this array is of type `struct rtwC_API_SampleTimeMap`. The members of this structure provide information about the sample period, offset, and whether or not the data is frame-based or sample-based.

Generate Example C API Files

Subtopics “C API Signals” on page 28-11, “C API States” on page 28-14, “C API Root-Level Inputs and Outputs” on page 28-15, and “C API Parameters” on page

28-16 discuss generated C API structures using the example model `rtwdemo_capi`. To generate code from the example model, do the following:

- 1 Open the model by clicking the `rtwdemo_capi` link above or by typing `rtwdemo_capi` on the MATLAB command line.
- 2 If you want to generate C API structures for root-level inputs/outputs in `rtwdemo_capi`, open the Configuration Parameters dialog box, go to the **Code Generation** > **Interface** pane, and select **Generate C API for: root-level I/O**.

Note: The setting of **Generate C API for: root-level I/O** must match between the top model and the referenced model. If you modify the option, save the top model and the referenced model to the same writable work folder.

- 3 Generate code for the model by double-clicking **Generate Code Using Simulink Coder**.

Note: The C API code examples in the next subtopics are generated with C as the target language.

This model has three global block output signals that will appear in C API generated code:

- `top_sig1`, which is a test point at the output of the Gain1 block in the top model
- `sig2_eg`, which appears in the top model and is defined in the base workspace as a `Simulink.Signal` object having storage class `ExportedGlobal`
- `bot_sig1`, which appears in the referenced model `rtwdemo_capi_bot` and is defined as a `Simulink.Signal` object having storage class `SimulinkGlobal`

The model also has two discrete states that will appear in the C API generated code:

- `top_state`, which is defined for the Delay1 block in the top model
- `bot_state`, which is defined for the Discrete Filter block in the referenced model

The model has root-level inputs/outputs that will appear in the C API generated code if you select the option **Generate C API for: root-level I/O**:

- Four root-level inputs, `In1` through `In4`
- Six root-level outputs, `Out1` through `Out6`

Additionally, the model has five global block parameters that will appear in C API generated code:

- Kp (top model Gain1 block and referenced model Gain2 block share)
- Ki (referenced model Gain3 block)
- p1 (lookup table lu1d)
- p2 (lookup table lu2d)
- p3 (lookup table lu3d)

C API Signals

The `rtwCAPI_Signals` structure captures signal information including the signal name, address, block path, output port number, data type information, dimensions information, fixed-point information, and sample-time information.

Here is the section of code in `rtwdemo_capi_capi.c` that provides information on C API signals for the top model in `rtwdemo_capi`:

```
/* Block output signal information */
static const rtwCAPI_Signals rtBlockSignals[] = {
    /* addrMapIndex, sysNum, blockPath,
     * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
     */
    { 0, 0, "rtwdemo_capi/Gain1",
      "top_sig1", 0, 0, 0, 0, 0 },

    { 1, 0, "rtwdemo_capi/lu2d",
      "sig2_eg", 0, 0, 1, 0, 0 },

    {
      0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
    }
};
```

Note To better understand the code, read the comments in the file. For example, notice the comment that begins on the third line in the preceding code. This comment lists the members of the `rtwCAPI_Signals` structure, in order. This tells you the order in which the assigned values for each member appear for a signal. In this example, the comment tells you that `signalName` is the fourth member of the structure. The following lines describe the first signal:

```
{ 0, 0, "rtwdemo_capi/Gain1",
  "top_sig1", 0, 0, 0, 0, 0 },
```

From these lines you infer that the name of the first signal is `top_sig1`.

Each array element, except the last, describes one output port for a block signal. The final array element is a sentinel, with all elements set to null values. For example, examine the second signal, described by the following code:

```
{ 1, 0, "rtwdemo_capi/lu2d",
  "sig2_eg", 0, 0, 1, 0, 0 },
```

This signal, named `sig2_eg`, is the output signal of the first port of the block `rtwdemo_capi/lu2d`. (This port is the first port because the zero-based index for `portNumber` displayed on the second line is assigned the value 0.)

The address of this signal is given by `addrMapIndex`, which, in this example, is displayed on the first line as 1. This provides an index into the `rtDataAddrMap` array, found later in `rtwdemo_capi_capi.c`:

```
/* Declare Data Addresses statically */
static void* rtDataAddrMap[] = {
    &rtwdemo_capi_B.top_sig1,          /* 0: Signal */
    &sig2_eg[0],                      /* 1: Signal */
    &rtwdemo_capi_DWork.top_state,    /* 2: Discrete State */
    &rtP_Ki,                          /* 3: Model Parameter */
    &rtP_Kp,                          /* 4: Model Parameter */
    &rtP_p1[0],                       /* 5: Model Parameter */
    &rtP_p2[0],                       /* 6: Model Parameter */
    &rtP_p3[0],                       /* 7: Model Parameter */
};
```

The index of 1 points to the second element in the `rtDataAddrMap` array. From the `rtDataAddrMap` array, you can infer that the address of this signal is `&sig2_eg[0]`.

This level of indirection supports multiple code instances of the same model. For multiple instances, the signal information remains constant, except for the address. In this case, the model is a single instance. Therefore, the `rtDataAddrMap` is declared statically. If you choose to generate reusable code, an initialize function is generated that initializes the addresses dynamically per instance. (For details on generating reusable code, see “Entry-Point Functions and Scheduling” on page 15-2 and, in the Embedded Coder documentation, “Set Up Support for Code Reuse”.)

The `dataTypeIndex` provides an index into the `rtDataTypeMap` array, found later in `rtwdemo_capi_capi.c`, indicating the data type of the signal:

```
/* Data Type Map - use dataTypeMapIndex to access this structure */
static const rtwCAPI_DataTypeMap rtDataTypeMap[] = {
    /* cName, mwName, numElements, elemMapIndex, dataSize, slDataId, *
     * isComplex, isPointer */
    { "double", "real_T", 0, 0, sizeof(real_T), SS_DOUBLE, 0, 0 }
};
```

Because the index is 0 for `sig2_eg`, the index points to the first structure element in the array. You can infer that the data type of the signal is `double`. The value of `isComplex` is 0, indicating that the signal is not complex. Rather than providing the data type information directly in the `rtwCAPI_Signals` structure, a level of indirection is introduced. The indirection allows multiple signals that share the same data type to point to one map structure, saving memory for each signal.

The `dimIndex` (dimensions index) provides an index into the `rtDimensionMap` array, found later in `rtwdemo_capi_capi.c`, indicating the dimensions of the signal. Because this index is 1 for `sig2_eg`, the index points to the second element in the `rtDimensionMap` array:

```
/* Dimension Map - use dimensionMapIndex to access elements of ths structure*/
static const rtwCAPI_DimensionMap rtDimensionMap[] = {
    /* dataOrientation, dimArrayIndex, numDims, vardimsIndex */
    { rtwCAPI_SCALAR, 0, 2, 0 },

    { rtwCAPI_VECTOR, 2, 2, 0 },
    ...
};
```

From this structure, you can infer that this is a nonscalar signal having a dimension of 2. The `dimArrayIndex` value, 2, provides an index into `rtDimensionArray`, found later in `rtwdemo_capi_capi.c`:

```
/* Dimension Array- use dimArrayIndex to access elements of this array */
static const uint_T rtDimensionArray[] = {
    1,          /* 0 */
    1,          /* 1 */
    2,          /* 2 */
    ...
};
```

The `fxpIndex` (fixed-point index) provides an index into the `rtFixPtMap` array, found later in `rtwdemo_capi_capi.c`, indicating fixed-point information about the signal. Your code can use the scaling information to compute the real-world value of the signal, using the equation $V = SQ + B$, where V is “real-world” (that is, base-10) value, S is user-specified slope, Q is “quantized fixed-point value” or “stored integer,” and B is user-specified bias. (For details, see “Scaling” in the Fixed-Point Designer documentation.)

Because this index is 0 for `sig2_eg`, the signal does not have fixed-point information. A fixed-point map index of zero means that the signal does not have fixed-point information.

The `sTimeIndex` (sample-time index) provides the index to the `rtSampleTimeMap` array, found later in `rtwdemo_capi_capi.c`, indicating task information about the

signal. If you log multirate signals or conditionally executed signals, the sampling information can be useful.

Note: `model_capi.c` (or `.cpp`) includes `rtw_capi.h`. A source file that references the `rtBlockSignals` array also must include `rtw_capi.h`.

C API States

The `rtwCAPI_States` structure captures state information including the state name, address, block path, type (continuous or discrete), data type information, dimensions information, fixed-point information, and sample-time information.

Here is the section of code in `rtwdemo_capi_capi.c` that provides information on C API states for the top model in `rtwdemo_capi`:

```
/* Block states information */
static const rtwCAPI_States rtBlockStates[] = {
    /* addrMapIndex, contStateStartIndex, blockPath,
     * stateName, pathAlias, dWorkIndex, dataTypeIndex, dimIndex,
     * fixPtIdx, sTimeIndex, isContinuous
     */
    { 2, -1, "rtwdemo_capi/Delay1",
      "top_state", "", 0, 0, 0, 0, 0, 0 },

    {
    0, -1, (NULL), (NULL), (NULL), 0, 0, 0, 0, 0, 0
    }
};
```

Each array element, except the last, describes a state in the model. The final array element is a sentinel, with all elements set to null values. In this example, the C API code for the top model displays one state:

```
{ 2, -1, "rtwdemo_capi/Delay1",
  "top_state", "", 0, 0, 0, 0, 0, 0 },
```

This state, named `top_state`, is defined for the block `rtwdemo_capi/Delay1`. The value of `isContinuous` is zero, indicating that the state is discrete rather than continuous. The other fields correspond to the like-named signal equivalents described in “C API Signals” on page 28-11, as follows:

- The address of the signal is given by `addrMapIndex`, which, in this example, is 2. This is an index into the `rtDataAddrMap` array, found later in

`rtwdemo_capi_capi.c`. Because the index is zero based, 2 corresponds to the third element in `rtDataAddrMap`, which is `&rtwdemo_capi_DWork.top_state`.

- The `dataTypeIndex` provides an index into the `rtDataTypeMap` array, found later in `rtwdemo_capi_capi.c`, indicating the data type of the parameter. The value 0 corresponds to a double, noncomplex parameter.
- The `dimIndex` (dimensions index) provides an index into the `rtDimensionMap` array, found later in `rtwdemo_capi_capi.c`. The value 0 corresponds to the first entry, which is `{ rtwCAPI_SCALAR, 0, 2, 0 }`.
- The `fixPtIndex` (fixed-point index) provides an index into the `rtFixPtMap` array, found later in `rtwdemo_capi_capi.c`, indicating fixed-point information about the parameter. As with the corresponding signal attribute, a fixed-point map index of zero means that the parameter does not have fixed-point information.

C API Root-Level Inputs and Outputs

The `rtwCAPI_Signals` structure captures root-level input/output information including the input/output name, address, block path, port number, data type information, dimensions information, fixed-point information, and sample-time information. (This structure also is used for block output signals, as previously described in “C API Signals” on page 28-11.)

Here is the section of code in `rtwdemo_capi_capi.c` that provides information on C API root-level inputs/outputs for the top model in `rtwdemo_capi`:

```
/* Root Inputs information */
static const rtwCAPI_Signals rtRootInputs[] = {
    /* addrMapIndex, sysNum, blockPath,
     * signalName, portNumber, dataTypeIndex, dimIndex, fixpIndex, sTimeIndex
     */
    { 3, 0, "rtwdemo_capi/In1",
      "", 1, 0, 0, 0, 0, 0 },

    { 4, 0, "rtwdemo_capi/In2",
      "", 2, 0, 0, 0, 0, 0 },

    { 5, 0, "rtwdemo_capi/In3",
      "", 3, 0, 0, 0, 0, 0 },

    { 6, 0, "rtwdemo_capi/In4",
      "", 4, 0, 0, 0, 0, 0 },

    {
      0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
    }
};
```

```

/* Root Outputs information */
static const rtwCAPI_Signals rtRootOutputs[] = {
/* addrMapIndex, sysNum, blockPath,
 * signalName, portNumber, dataTypeIndex, dimIndex, fxpIndex, sTimeIndex
 */
{ 7, 0, "rtwdemo_capi/Out1",
  "", 1, 0, 0, 0, 0 },

{ 8, 0, "rtwdemo_capi/Out2",
  "", 2, 0, 0, 0, 0 },

{ 9, 0, "rtwdemo_capi/Out3",
  "", 3, 0, 0, 0, 0 },

{ 10, 0, "rtwdemo_capi/Out4",
  "", 4, 0, 0, 0, 0 },

{ 11, 0, "rtwdemo_capi/Out5",
  "sig2_eg", 5, 0, 1, 0, 0 },

{ 12, 0, "rtwdemo_capi/Out6",
  "", 6, 0, 1, 0, 0 },

{
  0, 0, (NULL), (NULL), 0, 0, 0, 0, 0
}
};

```

For information about interpreting the values in the `rtwCAPI_Signals` structure, see the previous section “C API Signals” on page 28-11.

C API Parameters

The `rtwCAPI_BlockParameters` and `rtwCAPI_ModelParameters` structures capture parameter information including the parameter name, block path (for block parameters), address, data type information, dimensions information, and fixed-point information.

The `rtModelParameters` array contains entries for workspace variables that are referenced as tunable Simulink block parameters or Stateflow data of machine scope. For example, tunable parameters include `Simulink.Parameter` objects that use a storage class other than `Auto`. The Simulink Coder software assigns its elements only `NULL` or zero values in the absence of such data.

The setting that you choose for **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** determines how information is generated into the `rtBlockParameters` array in `model_capi.c` (or `.cpp`).

- If you set **Default parameter behavior** to `Tunable`, the `rtBlockParameters` array contains an entry for every modifiable parameter of every block in the model.

However, if you use a MATLAB variable or a tunable parameter to specify a block parameter, the block parameter does not appear in `rtBlockParameters`. Instead, the variable or tunable parameter appears in `rtModelParameters`.

- If you set **Default parameter behavior** to **Inlined**, the `rtBlockParameters` array is empty. The Simulink Coder software assigns its elements only `NULL` or zero values.

The last member of each array is a sentinel, with all elements set to null values.

Here is the `rtBlockParameters` array that is generated by default in `rtwdemo_capi_capi.c`:

```
/* Individual block tuning is not valid when inline parameters is *
 * selected. An empty map is produced to provide a consistent *
 * interface independent of inlining parameters. *
 */
static const rtwCAPI_BlockParameters rtBlockParameters[] = {
    /* addrMapIndex, blockPath,
     * paramName, dataTypeIndex, dimIndex, fixPtIdx
     */
    {
        0, (NULL), (NULL), 0, 0, 0
    }
};
```

In this example, only the final, sentinel array element is generated, with all members of the structure `rtwCAPI_BlockParameters` set to `NULL` and zero values. This is because **Default parameter behavior** is set to **Inlined** by default for the `rtwdemo_capi` example model. If you set **Default parameter behavior** to **Tunable**, the block parameters are generated in the `rtwCAPI_BlockParameters` structure. However, MATLAB variables and tunable parameters appear in the `rtwCAPI_ModelParameters` structure.

Here is the `rtModelParameters` array that is generated by default in `rtwdemo_capi_capi.c`:

```
/* Tunable variable parameters */
static const rtwCAPI_ModelParameters rtModelParameters[] = {
    /* addrMapIndex, varName, dataTypeIndex, dimIndex, fixPtIndex */
    { 2, TARGET_STRING("Ki"), 0, 0, 0 },

    { 3, TARGET_STRING("Kp"), 0, 0, 0 },

    { 4, TARGET_STRING("p1"), 0, 2, 0 },

    { 5, TARGET_STRING("p2"), 0, 3, 0 },
};
```

```
{ 6, TARGET_STRING("p3"), 0, 4, 0 },  
{ 0, (NULL), 0, 0, 0 }  
};
```

In this example, the `rtModelParameters` array contains entries for each variable that is referenced as a tunable Simulink block parameter.

For example, the `varName` (variable name) of the fourth parameter is `p2`. The other fields correspond to the like-named signal equivalents described in “C API Signals” on page 28-11, as follows:

- The address of the fourth parameter is given by `addrMapIndex`, which, in this example, is 5. This is an index into the `rtDataAddrMap` array, found later in `rtwdemo_capi_capi.c`. Because the index is zero based, 5 corresponds to the sixth element in `rtDataAddrMap`, which is `rtP_p2`.
- The `dataTypeIndex` provides an index into the `rtDataTypeMap` array, found later in `rtwdemo_capi_capi.c`, indicating the data type of the parameter. The value 0 corresponds to a double, noncomplex parameter.
- The `dimIndex` (dimensions index) provides an index into the `rtDimensionMap` array, found later in `rtwdemo_capi_capi.c`. The value 3 corresponds to the fourth entry, which is `{ rtwCAPI_MATRIX_COL_MAJOR, 6, 2, 0 }`.
- The `fixPtIndex` (fixed-point index) provides an index into the `rtFixPtMap` array, found later in `rtwdemo_capi_capi.c`, indicating fixed-point information about the parameter. As with the corresponding signal attribute, a fixed-point map index of zero means that the parameter does not have fixed-point information.

For more information about tunable parameter storage in the generated code, see “Block Parameter Representation in the Generated Code” on page 14-43.

Map C API Data Structures to `rtModel`

The real-time model data structure encapsulates model data and associated information that describes the model fully. When you select the C API feature and generate code, the Simulink Coder code generator adds another member to the real-time model data structure generated in `model.h`:

```
/*  
 * DataMapInfo:  
 * The following substructure contains information regarding  
 * structures generated in the model's C API.
```

```

*/
struct {
    rtwCAPI_ModelMappingInfo mmi;
} DataMapInfo;

```

This member defines `mmi` (for model mapping information) of type `struct rtwCAPI_ModelMappingInfo`. The structure is located in `matlabroot/rtw/c/src/rtw_modelmap.h`. The `mmi` substructure defines the interface between the model and the C API files. More specifically, members of `mmi` map the real-time model data structure to the structures in `model_capi.c` (or `.cpp`).

Initializing values of `mmi` members to the arrays accomplishes the mapping, as shown in Map Model to C API Arrays of Structures. Each member points to one of the arrays of structures in the generated C API file. For example, the address of the `rtBlockSignals` array of structures is allocated to the first member of the `mmi` substructure in `model.c` (or `.cpp`), using the following code in the `rtw_modelmap.h` file:

```

/* signals */
struct {
    rtwCAPI_Signals const *signals;    /* Signals Array */
    uint_T                numSignals; /* Num Signals */
    rtwCAPI_Signals const *rootInputs; /* Root Inputs array */
    uint_T                numRootInputs; /* Num Root Inputs */
    rtwCAPI_Signals const *rootOutputs; /* Root Outputs array */
    uint_T                numRootOutputs; /* Num Root Outputs */
} Signals;

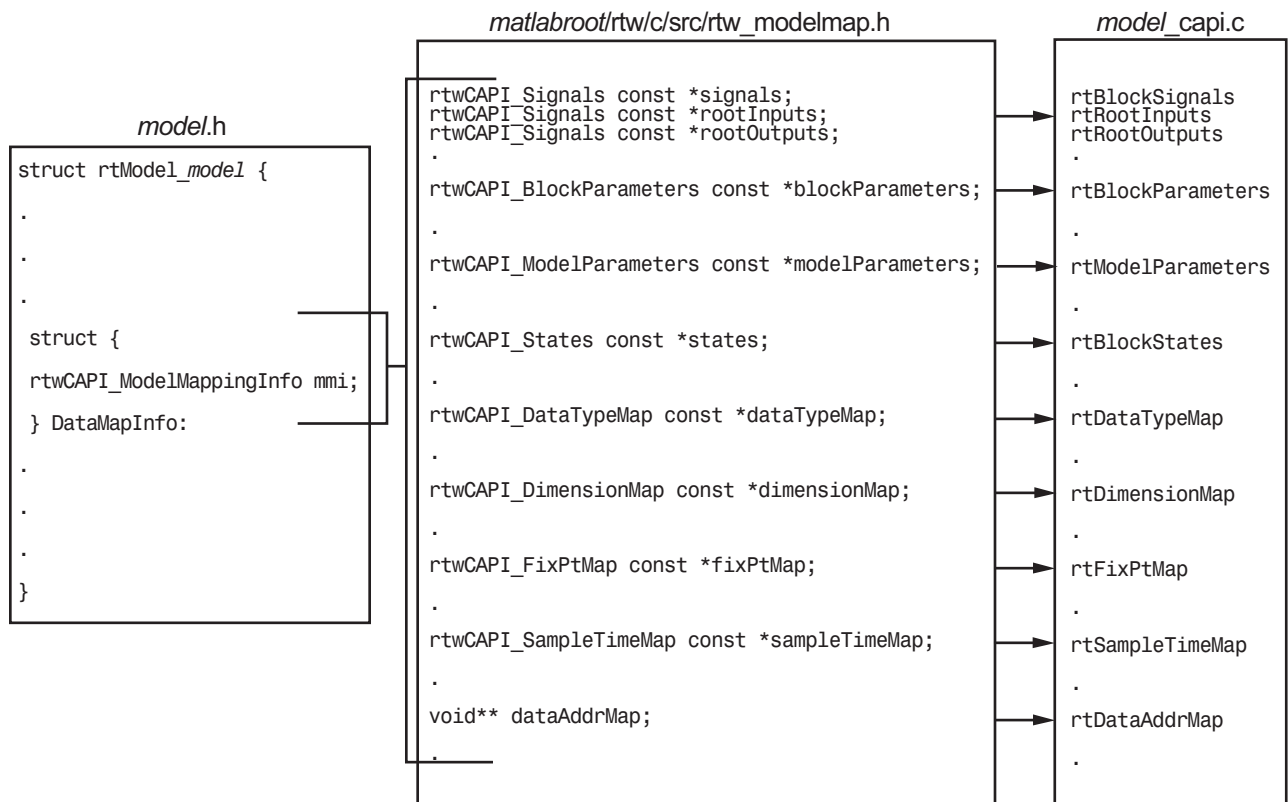
```

The model initialize function in `model.c` (or `.cpp`) performs the initializing by calling the C API initialize function. For example, the following code is generated in the model initialize function for example model `rtwdemo_capi`:

```

/* Initialize DataMapInfo substructure containing ModelMap for C API */
rtwdemo_capi_InitializeDataMapInfo(rtwdemo_capi_M);

```



Map Model to C API Arrays of Structures

Note: This figure lists the arrays in the order that their structures appear in `rtw_modelmap.h`, which differs slightly from their generated order in `model_capi.c`.

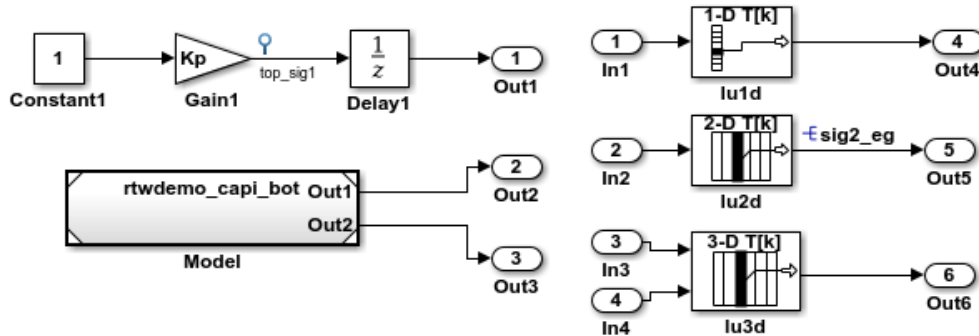
Generate C API Data Definition File for Exchanging Data with a Target System

This model illustrates the target-based C API for interfacing signals, parameters, and states in the generated code.

Open Example Model

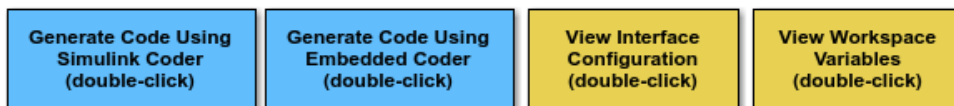
Open the example model `rtwdemo_capi`.

```
open_system('rtwdemo_capi');
```



This model illustrates the target-based C API for interfacing signals, parameters and states in the generated code. The C API is useful for real-time interaction with the data, without having to stop execution or recompile the generated code. Typically, a client/server protocol is set up from a host to a target using serial, TCP/IP, or dual-port memory connection. The purpose of this example is not the client/server protocol. Rather, this model shows the necessary data interface required by the C client/server programs.

You can enable the C API by selecting one or more C API options on the "Code Generation > Interface" pane of the Configuration Parameters dialog box. Any signal or parameter or state with an addressable storage class is placed in the C API data structure in `<model>_capi.c`. Note that signals, states and parameters in the referenced model can be accessed using C-API. So make sure that C-API is enabled for the referenced model.



Copyright 1994-2012 The MathWorks, Inc.

The C API is useful for real-time interaction with application data, without having to stop execution or recompile the generated code. Typically, a client/server protocol is set up from a host to a target using serial, TCP/IP, or dual-port memory connection. The

purpose of this example is not the client/server protocol. Rather, this model shows the necessary data interface required by the C client/server programs.

You enable the C API by selecting one or more C API options on the **Code Generation > Interface** pane of the Configuration Parameters dialog box. Any signal or parameter or state with an addressable storage class is placed in the C API data structure in *model_capi.c*. Note that signals, states, and parameters in the referenced model can be accessed using C API. So make sure that C API is enabled for the referenced model.

C API Limitations

The C API feature has the following limitations.

- The C API does not support the following values for the **CodeFormat** TLC variable:
 - **S-Function**
 - **Accelerator_S-Function** (for accelerated simulation)
- For ERT-based targets, the C API requires that support for floating-point code be enabled.
- Local block output signals are not supported.
- Local Stateflow parameters are not supported.
- The following custom storage class objects are not supported:
 - Objects without the package `csc_registration` file
 - Grouped custom storage classes
 - Objects defined by using macros
 - **BitField** objects
 - **FileScope** objects
- Customized data placement is disabled when you are using the C API. The interface looks for global data declaration in *model.h* and *model_private.h*. Declarations placed in any other file by customized data placement result in code that does not compile.

Note Custom Storage Class objects work in code generation, only if you use the ERT target and clear the **Ignore custom storage classes** check box in the Configuration Parameters dialog box.

Related Examples

- “Access Signal, State, and Parameter Data During Execution” on page 14-3

Use C API to Access Model Signals and States

This example helps you get started writing application code to interact with model signals and states. To get started writing application code to interact with model parameters, see “Use C API to Access Model Parameters” on page 28-30.

The C API provides you with the flexibility of writing your own application code to interact with model signals, states, root-level inputs/outputs, and parameters. Your target-based application code is compiled with the Simulink Coder generated code into an executable. The target-based application code accesses the C API structure arrays in *model_capi.c* (or *.cpp*). You might have host-based code that interacts with your target-based application code. Or, you might have other target-based code that interacts with your target-based application code. The files *rtw_modelmap.h* and *rtw_capi.h*, located in *matlabroot/rtw/c/src* (open), provide macros for accessing the structures in these arrays and their members.

Here is an example application that logs global signals and states in a model to a text file. This code is intended as a starting point for accessing signal and state addresses. You can extend the code to perform signal logging and monitoring, state logging and monitoring, or both.

This example uses the following macro and function interfaces:

- `rtmGetDataMapInfo` macro

Accesses the model mapping information (MMI) substructure of the real-time model structure. In the following macro call, `rtM` is the pointer to the real-time model structure in *model.c* (or *.cpp*):

```
rtwCAPI_ModelMappingInfo* mmi = &(rtmGetDataMapInfo(rtM).mmi);
```

- `rtmGetTPtr` macro

Accesses the absolute time information for the base rate from the timing substructure of the real-time model structure. In the following macro call, `rtM` is the pointer to the real-time model structure in *model.c* (or *.cpp*):

```
rtmGetTPtr(rtM)
```

- Custom functions `capi_StartLogging`, `capi_UpdateLogging`, and `capi_TerminateLogging`, provided via the files *rtwdemo_capi_datalog.h* and *rtwdemo_capi_datalog.c*. These files are located in *matlabroot/toolbox/rtw/rtwdemos* (open).

- `capi_StartLogging` initializes signal and state logging.
- `capi_UpdateLogging` logs a signal and state value at each time step.
- `capi_TerminateLogging` terminates signal and state logging and writes the logged values to a text file.

You can integrate these custom functions into generated model code using one or more of the following methods:

- **Code Generation > Custom Code** pane of the Configuration Parameters dialog box
- Custom Code library blocks
- TLC custom code functions

This tutorial uses the **Code Generation > Custom Code** pane and the System Outputs block from the Custom Code library to insert calls to the custom functions into `model.c` (or `.cpp`), as follows:

- `capi_StartLogging` is called in the `model_initialize` function.
- `capi_UpdateLogging` is called in the `model_step` function.
- `capi_TerminateLogging` is called in the `model_terminate` function.

The following excerpts of generated code from `model.c` (rearranged to reflect their order of execution) show how the function interfaces are used.

```
void rtwdemo_capi_initialize(void)
{
...
/* user code (Initialize function Body) */

/* C API Custom Logging Function: Start Signal and State logging via C API.
 * capi_StartLogging: Function prototype in rtwdemo_capi_datalog.h
 */
{
rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
printf("*** Started state/signal logging via C API **\n");
capi_StartLogging(MMI, MAX_DATA_POINTS);
}
...
}
...
/* Model step function */
void rtwdemo_capi_step(void)
{
...
/* user code (Output function Trailer) */

/* System '<Root>' */
```

```

/* C API Custom Logging Function: Update Signal and State logging buffers.
 * capi_UpdateLogging: Function prototype in rtwdemo_capi_datalog.h
 */
{
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
    capi_UpdateLogging(MMI, rtmGetTPtr(rtwdemo_capi_M));
}
...
}
...
/* Model terminate function */
void rtwdemo_capi_terminate(void)
{
    /* user code (Terminate function Body) */

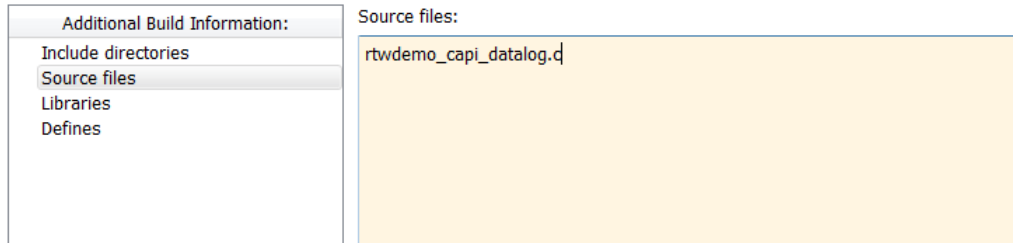
    /* C API Custom Logging Function: Dump Signal and State buffers into a text file.
     * capi_TerminateLogging: Function prototype in rtwdemo_capi_datalog.h
     */
    {
        capi_TerminateLogging("rtwdemo_capi_ModelLog.txt");
        printf("** Finished state/signal logging. Created rtwdemo_capi_ModelLog.txt **\n");
    }
}

```

The following procedure illustrates how you can use the C API macro and function interfaces to log global signals and states in a model to a text file.

- 1 At the MATLAB command line, enter `rtwdemo_capi` to open the example model.
- 2 Save the top model `rtwdemo_capi` and the referenced model `rtwdemo_capi_bot` to the same writable work folder.
- 3 Open the Configuration Parameters dialog box.
- 4 If you are licensed for Embedded Coder software and you want to use the `ert.tlc` target instead of the default `grt.tlc`, go to the **Code Generation** pane and use the **System target file** field to select an `ert.tlc` target. Make sure that you also select `ert.tlc` for the referenced model `rtwdemo_capi_bot`.
- 5 In the top model, go to the **Code Generation > Interface** pane.
 - a In the **Data exchange interface** subgroup, verify that the model options **Generate C API for: signals** and **Generate C API for: states** are selected. This example also leaves **Generate C API for: parameters** selected.
 - b If you are using the `ert.tlc` system target file, verify that the option **Support: complex numbers** is selected.
 - c Click the **All Parameters** tab, and select the **MAT-file logging** option.
 - d Click **Apply**.
 - e Update configuration parameter settings in the referenced model, `rtwdemo_capi_bot`, to match changes you made in the top model.

- 6 Use the **Custom Code** pane to embed your custom application code in the generated code. Select the **Custom Code** pane, and then click **Include directories**. The **Include directories** input field is displayed.
- 7 In the **Include directories** field, type `matlabroot/toolbox/rtw/rtwdemos`, where `matlabroot` represents the root of your MATLAB installation folder. (If you are specifying a Windows path that contains a space, place the text inside double quotes.)
- 8 In the **Additional Build Information** subpane, click **Source files**, and type `rtwdemo_capi_datalog.c`.



- 9 In the **Include custom C code in generated** subpane, click **Source file**, and type or copy and paste the following include statement:

```
#include "rtwdemo_capi_datalog.h"
```

- 10 In the **Initialize function** field, type or copy and paste the following application code:

```
/* C API Custom Logging Function: Start Signal and State logging via C API.
 * capi_StartLogging: Function prototype in rtwdemo_capi_datalog.h
 */
{
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
    printf("*** Started state/signal logging via C API **\n");
    capi_StartLogging(MMI, MAX_DATA_POINTS);
}
```

Note: If you renamed the top model `rtwdemo_capi`, update the name `rtwdemo_capi_M` in the application code to reflect the new model name.

- 11 In the **Terminate function** field, type or copy and paste the following application code:

```
/* C API Custom Logging Function: Dump Signal and State buffers into a text file.
 * capi_TerminateLogging: Function prototype in rtwdemo_capi_datalog.h
 */
```

```

{
capi_TerminateLogging("rtwdemo_capi_ModelLog.txt");
printf("** Finished state/signal logging. Created rtwdemo_capi_ModelLog.txt **\n");
}

```

Click **Apply**.

- 12 In the MATLAB Command Window, enter `custcode` to open the Simulink Coder Custom Code library. At the top level of the `rtwdemo_capi` model, add a System Outputs block.
- 13 Double-click the System Outputs block to open the System Outputs Function Custom Code dialog box. In the **System Outputs Function Exit Code** field, type or copy and paste the following application code:

```

/* C API Custom Logging Function: Update Signal and State logging buffers.
 * capi_UpdateLogging: Function prototype in rtwdemo_capi_dataLog.h
 */
{
    rtwCAPI_ModelMappingInfo *MMI = &(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);
    capi_UpdateLogging(MMI, rtmGetTPtr(rtwdemo_capi_M));
}

```

Note: If you renamed the top model `rtwdemo_capi`, update two instances of the name `rtwdemo_capi_M` in the application code to reflect the new model name.

Click **OK**.

- 14 On the **Code Generation** pane, verify that the **Generate code only** check box is *cleared*.

Press **Ctrl+B** to build the model and generate an executable file. For example, on a Windows system, the build generates the executable file `rtwdemo_capi.exe` in your current working folder.

- 15 In the MATLAB Command Window, enter the command `!rtwdemo_capi` to run the executable file. During execution, signals and states are logged using the C API and then written to the text file `rtwdemo_capi_ModelLog.txt` in your current working folder.

```

>> !rtwdemo_capi

** starting the model **
** Started state/signal logging via C API **
** Logging 2 signal(s) and 1 state(s). In this demo, only scalar named
    signals/states are logged **
** Finished state/signal logging. Created rtwdemo_capi_ModelLog.txt **

```

- 16** Examine the text file in the MATLAB editor or other text editor. Here is an excerpt of the signal and state logging output.

```
***** Signal Log File *****
```

```
Number of Signals Logged: 2
```

```
Number of points (time steps) logged: 51
```

Time	bot_sig1 (Referenced Model)	top_sig1
0	70	4
0.2	70	4
0.4	70	4
0.6	70	4
0.8	70	4
1	70	4
1.2	70	4
1.4	70	4
1.6	70	4
1.8	70	4
2	70	4
...		

```
***** State Log File *****
```

```
Number of States Logged: 1
```

```
Number of points (time steps) logged: 51
```

Time	bot_state (Referenced Model)
0	0
0.2	70
0.4	35
0.6	52.5
0.8	43.75
1	48.13
1.2	45.94
1.4	47.03
1.6	46.48
1.8	46.76
2	46.62
...	

Use C API to Access Model Parameters

This example helps you get started writing application code to interact with model parameters. To get started writing application code to interact with model signals and states, see “Use C API to Access Model Signals and States” on page 28-24.

The C API provides you with the flexibility of writing your own application code to interact with model signals, states, root-level inputs/outputs, and parameters. Your target-based application code is compiled with the Simulink Coder generated code into an executable. The target-based application code accesses the C API structure arrays in *model_capi.c* (or *.cpp*). You might have host-based code that interacts with your target-based application code. Or, you might have other target-based code that interacts with your target-based application code. The files *rtw_modelmap.h* and *rtw_capi.h*, located in *matlabroot/rtw/c/src* (open), provide macros for accessing the structures in these arrays and their members.

Here is an example application that prints the parameter values of tunable parameters in a model to the standard output. This code is intended as a starting point for accessing parameter addresses. You can extend the code to perform parameter tuning. The application:

- Uses the `rtmGetDataMapInfo` macro to access the mapping information in the `mmi` substructure of the real-time model structure

```
rtwCAPI_ModelMappingInfo* mmi = &(rtmGetDataMapInfo(rtM).mmi);
```

where `rtM` is the pointer to the real-time model structure in *model.c* (or *.cpp*).

- Uses `rtwCAPI_GetNumModelParameters` to get the number of model parameters in mapped C API:

```
uint_T nModelParams = rtwCAPI_GetNumModelParameters(mmi);
```

- Uses `rtwCAPI_GetModelParameters` to access the array of model parameter structures mapped in C API:

```
rtwCAPI_ModelParameters* capiModelParams = \  
    rtwCAPI_GetModelParameters(mmi);
```

- Loops over the `capiModelParams` array to access individual parameter structures. A call to the function `capi_PrintModelParameter` displays the value of the parameter.

The example application code is provided below:


```

{
/* Get CAPI Mapping structure from Real-Time Model structure */
rtwCAPI_ModelMappingInfo* capiMap = \
&(rtmGetDataMapInfo(rtwdemo_capi_M).mmi);

/* Get number of Model Parameters from capiMap */
uint_T nModelParams = rtwCAPI_GetNumModelParameters(capiMap);
printf("Number of Model Parameters: %d\n", nModelParams);

/* If the model has Model Parameters, print them using the
application capi_PrintModelParameter */
if (nModelParams == 0) {
    printf("No Tunable Model Parameters in the model \n");
}
else {
    unsigned int idx;

    for (idx=0; idx < nModelParams; idx++) {
        /* call print utility function */
        capi_PrintModelParameter(capiMap, idx);
    }
}
}
}

```

The print utility function is located in *matlabroot/rtw/c/src/rtw_capi_examples.c*. This file contains utility functions for accessing the C API structures.

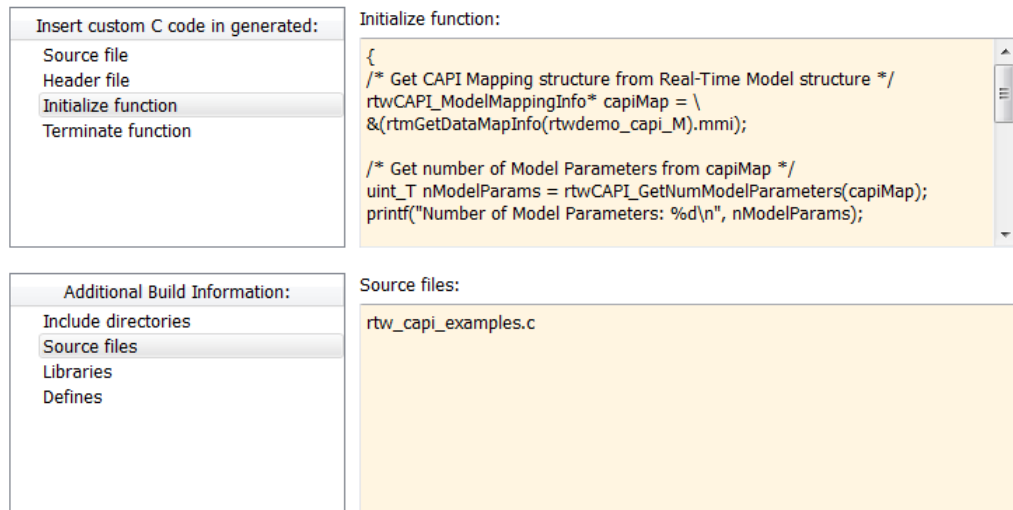
To become familiar with the example code, try building a model that displays the tunable block parameters and MATLAB variables. You can use *rtwdemo_capi*, the C API example model. The following steps apply to both *grt.tlc* and *ert.tlc* targets, unless otherwise indicated.

- 1 At the MATLAB command line, enter *rtwdemo_capi* to open the example model.
- 2 Save the top model *rtwdemo_capi* and the referenced model *rtwdemo_capi_bot* to the same writable work folder.
- 3 If you are licensed for Embedded Coder software and you want to use the *ert.tlc* target instead of the default *grt.tlc*, go to the **Code Generation** pane of the Configuration Parameters dialog box and use the **System target file** field to select an *ert.tlc* target. Make sure that you also select *ert.tlc* for the referenced model *rtwdemo_capi_bot*.
- 4 Go to the **Code Generation > Interface** pane.

- a In the **Data exchange interface** subgroup, verify that the model option **Generate C API for: parameters** is selected.
 - b If you are using the `ert.tlc` system target file, verify that the option **Support: complex numbers** is selected.
 - c Click the **All Parameters** tab, and select the **MAT-file logging** option.
 - d Click **Apply**.
 - e Update configuration parameter settings in the referenced model, `rtwdemo_capi_bot`, to match changes you made in the top model.
- 5 Use the **Custom Code** pane to embed your custom application code in the generated code. Select the **Custom Code** pane, and then click **Initialize function**. The **Initialize function** input field is displayed.
 - 6 In the **Initialize function** input field, type or copy and paste the example application code shown above step 1. This embeds the application code in the `model_initialize` function.

Note: If you renamed the top model `rtwdemo_capi`, update the name `rtwdemo_capi_M` in the application code to reflect the new model name.

- 7 Click **Include directories**, and type `matlabroot/rtw/c/src`, where *matlabroot* represents the root of your MATLAB installation folder. (If you are specifying a Windows path that contains a space, place the text inside double quotes.)
- 8 In the **Additional Build Information** subpane, click **Source files**, and type `rtw_capi_examples.c`.



Click **Apply**.

- On the **Code Generation** pane, verify that the **Generate code only** check box is *cleared*.

Press **Ctrl+B** to build the model and generate an executable file. For example, on a Windows system, the build generates the executable file `rtwdemo_capi.exe` in your current working folder.

- In the MATLAB Command Window, enter `!rtwdemo_capi` to run the executable file. Running the program displays parameter information in the Command Window.

```
>> !rtwdemo_capi

** starting the model **
Number of Model Parameters: 5
Ki =
    7
Kp =
    4
p1 =
    0.15
    0.36
    0.81
```

```
p2 =  
    0.09    0.75    0.57  
    0.13    0.96    0.059  
p3 =  
ans(:,:,1) =  
    0.23    0.82    0.04    0.64  
    0.35    0.01    0.16    0.73  
  
ans(:,:,2) =  
    0.64    0.54    0.74    0.68  
    0.45    0.29    0.18    0.18
```

ASAP2 Data Measurement and Calibration in Simulink Coder

Export ASAP2 File for Data Measurement and Calibration

The ASAM MCD-2 MC standard, also known as ASAP2, is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a non-object-oriented description of the data used for measurement, calibration, and diagnostic systems. For more information on ASAM and the ASAM MCD-2 MC (ASAP2) standard, see the ASAM Web site at <http://www.asam.net>.

The Simulink Coder product lets you export an ASAP2 file containing information about your model during the code generation process.

You can run an interactive example of ASAP2 file generation. To open the example at the MATLAB command prompt, enter the following command:

```
rtwdemo_asap2
```

Note: Simulink Coder support for ASAP2 file generation is version-neutral. By default, the software generates ASAP2 version 1.31 format, but the generated model information is generally compatible with all ASAP2 versions. ASAP2 file generation also is neutral with respect to the specific needs of ASAP2 measurement and calibration tools. The software provides customization APIs that you can use to customize ASAP2 file generation to generate any ASAP2 version and to meet the specific needs of your ASAP2 tools.

In this section...

“What You Should Know” on page 29-2

“Targets Supporting ASAP2” on page 29-3

“Define ASAP2 Information” on page 29-3

“Generate an ASAP2 File” on page 29-9

“Structure of the ASAP2 File” on page 29-12

“Create a Host-Based ASAM-ASAP2 Data Definition File for Data Measurement and Calibration” on page 29-13

What You Should Know

To make use of ASAP2 file generation, you should become familiar with the following topics:

- ASAM and the ASAP2 standard and terminology. See the ASAM Web site at <http://www.asam.net>.
- Simulink data objects. Data objects are used to supply information not contained in the model. For an overview, see “Data Objects”.
- Storage and representation of signals and parameters in generated code. See “Data Representation”.
- If you are licensed for Embedded Coder, see also the Embedded Coder topic “Data Representation”.

Targets Supporting ASAP2

ASAP2 file generation is available to all Simulink Coder target configurations. You can select these target configurations from the System Target File Browser. For example,

- The **Generic Real-Time Target (grt.tlc)** lets you generate an ASAP2 file as part of the code generation and build process.
- The **Embedded Coder (ert.tlc)** target selections also lets you generate an ASAP2 file as part of the code generation and build process.
- The **ASAM-ASAP2 Data Definition Target (asap2.tlc)** lets you generate only an ASAP2 file, without building an executable.

Procedures for generating ASAP2 files by using these target configurations are given in “Generate an ASAP2 File” on page 29-9.

Define ASAP2 Information

- “Define ASAP2 Information for Parameters and Signals” on page 29-3
- “Memory Address Attribute” on page 29-4
- “Automatic ECU Address Replacement for ASAP2 Files (Embedded Coder)” on page 29-6
- “Define ASAP2 Information for Lookup Tables” on page 29-7

Define ASAP2 Information for Parameters and Signals

The ASAP2 file generation process requires information about parameters and signals in your model. Some of this information is contained in the model itself. You must supply the rest by using Simulink data objects and corresponding properties.

You can use built-in Simulink data objects to provide the information. For example, you can use `Simulink.Signal` objects to provide MEASUREMENT information and `Simulink.Parameter` objects to provide CHARACTERISTIC information. Also, you can use data objects from data classes that are derived from `Simulink.Signal` and `Simulink.Parameter` to provide the information. For details, see “Data Objects”.

The following table contains the minimum set of data attributes required for ASAP2 file generation. Some data attributes are defined in the model; others are supplied in the properties of objects. For attributes that are defined in `Simulink.Signal` or `Simulink.Parameter` objects, the table gives the associated property name.

Data Attribute	Defined In	Property Name
Name (symbol)	Data object	Inherited from the handle of the data object to which parameter or signal name resolves
Description	Data object	Description
Data type	Model	Not applicable
Scaling (if fixed-point data type)	Model	Data type (for signals) Inherited from value (for parameters)
Minimum allowable value	Data object	Min
Maximum allowable value	Data object	Max
Unit	Data object	Unit
Memory address (optional)	Data object	MemoryAddress_ASAP2 (optional; see “Memory Address Attribute” on page 29-4.)

Memory Address Attribute

If the memory address attribute is unknown before code generation, the code generator inserts ECU Address placeholder text in the generated ASAP2 file. You can substitute an actual address for the placeholder by postprocessing the generated file. See the file `matlabroot/toolbox/rtw/targets/asap2/asap2/asap2post.m` for an example. `asap2post.m` parses through the linker map file that you provide and replaces the

ECU Address placeholders in the ASAP2 file with the actual memory addresses. Since linker map files vary from compiler to compiler, you might need to modify the regular expression code in `asap2post.m` to match the format of the linker map you use.

Note: If Embedded Coder is licensed and installed on your system, and if you are generating ELF (Executable and Linkable Format) files for your embedded target, you can use the `rtw.asap2SetAddress` function to automate ECU address replacement. For more information, see “Automatic ECU Address Replacement for ASAP2 Files (Embedded Coder)” on page 29-6.

If the memory address attribute is known before code generation, it can be defined in the data object. By default, the `MemoryAddress_ASAP2` property does not exist in the `Simulink.Signal` or `Simulink.Parameter` data object classes. If you want to add the attribute, add a property called `MemoryAddress_ASAP2` to a custom class that is a subclass of the `Simulink` or `ASAP2` class. For information on subclassing Simulink data classes, see “Define Data Classes” in the Simulink documentation.

Note In previous releases, for ASAP2 file generation, you had to define objects explicitly as `ASAP2.Signal` and `ASAP2.Parameter`. This is no longer a limitation. As explained above, you can use built-in Simulink objects for generating an ASAP2 file. If you have been using an earlier release, you can continue to use the ASAP2 objects. If one of these ASAP2 objects was created in the previous release, and you use it in this release, the MATLAB Command Window displays a warning the first time the objects are loaded.

The following table indicates the Simulink object properties that have replaced the ASAP2 object properties of the previous release:

Differences Between ASAP2 and Simulink Parameter and Signal Object Properties

ASAP2 Object Properties (Previous)	Simulink Object Properties (Current)
<code>LONGIG_ASAP2</code>	Description
<code>PhysicalMin_ASAP2</code>	Min
<code>PhysicalMax_ASAP2</code>	Max
<code>Units_ASAP2</code>	Unit

Automatic ECU Address Replacement for ASAP2 Files (Embedded Coder)

If Embedded Coder is licensed and installed on your system, and if you are generating ELF (Executable and Linkable Format) files for your embedded target, you can use the `rtw.asap2SetAddress` function to automate the replacement of ECU Address placeholder memory address values with actual addresses in the generated ASAP2 file.

If the memory address attribute is unknown before code generation, the code generator inserts ECU Address placeholder text in the generated ASAP2 file, as shown in the example below.

```
/begin CHARACTERISTIC
/* Name          */ Ki
/* Long Identifier */ ""
/* Type          */ VALUE
/* ECU Address   */ 0x0
                  /*ECU_Address@Ki@ */
```

To substitute actual addresses for the ECU Address placeholders, process the generated ASAP2 file using the `rtw.asap2SetAddress` function. The general syntax is as follows:

```
rtw.asap2SetAddress(ASAP2File, InfoFile)
```

where the arguments are character vectors specifying the name of the generated ASAP2 file and the name of the generated executable ELF file or DWARF debug information file for the model. When called, `rtw.asap2SetAddress` extracts the actual ECU address from the specified ELF or DWARF file and replaces the placeholder in the ASAP2 file with the actual address, for example:

```
/begin CHARACTERISTIC
/* Name          */ Ki
/* Long Identifier */ ""
/* Type          */ VALUE
/* ECU Address   */ 0x40009E60
```

Define ASAP2 Information for Lookup Tables

Simulink Coder software generates ASAP2 descriptions for lookup table data and its breakpoints. The software represents 1-D table data as **CURVE** information, 2-D table data as **MAP** information, and breakpoints as **AXIS_DESCR** and **AXIS_PTS** information. You can model lookup tables using one of the following Simulink Lookup Table blocks:

- Direct Lookup Table (n-D) — 1 and 2 dimensions
- Interpolation Using Prelookup — 1 and 2 dimensions
- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table — 1 and 2 dimensions

The software supports the following types of lookup table breakpoints (axis points):

Breakpoint Type	Generates
Tunable and shared among multiple table axes (common axis)	COM_AXIS
Fixed and nontunable (fixed axis)	One of these variants of FIX_AXIS: <ul style="list-style-type: none"> • FIX_AXIS_PAR if breakpoints are integers with equidistant spacing and the equidistant spacing is a power of two • FIX_AXIS_PAR_DIST if breakpoints are integers with equidistant spacing • FIX_AXIS_PAR_LIST if breakpoints are integers with non-equidistant spacing
Tunable but not shared among multiple tables (standard axis)	STD_AXIS

When you configure the blocks for ASAP2 code generation:

- For table data, use a `Simulink.Parameter` data object with a non-`Auto` storage class.
- For tunable breakpoint data that is shared among multiple table axes (`COM_AXIS`), use a `Simulink.Parameter` data object with a non-`Auto` storage class.
- For fixed, nontunable breakpoint data (`FIX_AXIS`), use workspace variables or arrays specified in the block parameters dialog box. The breakpoints should be stored as integers in the code, so the data type should be a built-in integer type (`int8`, `int16`, `int32`, `uint8`, `uint16`, or `uint32`), a fixed-point data type, or an equivalent alias type.
- For tunable breakpoint data that is not shared among multiple tables (`STD_AXIS`):
 - 1 Create a `Simulink.Bus` object to define the `struct` packaging (names and order of the fields). The fields of the parameter structure must correspond to the lookup table data and each axis of the lookup table block. For example, in an n-D Lookup Table block with 2 dimensions, the structure must contain only three fields. This bus object describes the record layout for the lookup characteristic.
 - 2 Create a `Simulink.Parameter` object to represent a tunable parameter.
 - 3 Create table and axis values.
 - 4 Optionally, specify the **Units**, **Minimum**, and **Maximum** properties for the parameter object. The properties will be applied to table data only.

Here is an example of an n-D Lookup Table record generated into an ASAP2 file in Standard Axis format:

```
/begin CHARACTERISTIC
  /* Name */   STDAxisParam
  ...
  /* Record Layout */   Lookup1D_X_WORD_Y_FLOAT32_IEEE
  ...
  begin AXIS_DESCR
    /* Description of X-Axis Points */
    /* Axis Type */   STD_AXIS
    ...
  /end AXIS_DESCR
/end CHARACTERISTIC

/begin RECORD_LAYOUT Lookup1D_X_WORD_Y_FLOAT32_IEEE
  AXIS_PTS_X 1 WORD INDEX_INCR DIRECT
```

```
FNC_VALUES 2 FLOAT32_IEEE COLUMN_DIR DIRECT
/end RECORD_LAYOUT
```

Note: The example model `rtwdemo_asap2` illustrates ASAP2 file generation for Lookup Table blocks, including both tunable (`COM_AXIS`) and fixed (`FIX_AXIS`) lookup table breakpoints.

Generate an ASAP2 File

- “About Generating ASAP2 Files” on page 29-9
- “Use GRT or ERT Target” on page 29-9
- “Use the ASAM-ASAP2 Data Definition Target” on page 29-10
- “Generate ASAP2 Files for Referenced Models” on page 29-11
- “Merge ASAP2 Files for Top and Referenced Models” on page 29-12

About Generating ASAP2 Files

You can generate an ASAP2 file from your model in one of the following ways:

- Use the Generic Real-Time Target or a Embedded Coder target to generate an ASAP2 file as part of the code generation and build process.
- Use the ASAM-ASAP2 Data Definition Target to generate only an ASAP2 file, without building an executable.

This section discusses how to generate an ASAP2 file by using the targets that have built-in ASAP2 support. For an example, see the ASAP2 example model `rtwdemo_asap2`.

Use GRT or ERT Target

The procedure for generating the ASAP2 data definition for a model using the Generic Real-Time Target or an Embedded Coder target is as follows:

- 1 Create the desired model. Use parameter names and signal labels to refer to corresponding `CHARACTERISTIC` records and `MEASUREMENT` records, respectively.
- 2 Create `Simulink.Parameter` and `Simulink.Signal` objects in the MATLAB workspace and reference them from parameters and signals in the model. A convenient way of creating multiple signal and parameter data objects is to use the Data Object Wizard. Alternatively, you can create data objects one at a time from

the MATLAB command line. For details on how to use the Data Object Wizard, see “Create Data Objects for a Model Using Data Object Wizard” in the Simulink documentation.

- 3 For each data object, configure the **Storage class** property to a setting other than **Auto**, **FileScope**, or **SimulinkGlobal**. This setting declares the data object as global in the generated code. For example, a storage class setting of **ExportedGlobal** configures the data object as unstructured global in the generated code.

Note: The data object is not represented in the ASAP2 file if any of the following conditions exist:

- You set the storage class to **Auto**, **FileScope**, or **SimulinkGlobal**.
- You set the storage class to **Custom** and custom storage class settings cause the code generator to generate a macro or non-addressable variable.

-
- 4 Configure the remaining properties as desired for each data object.
 - 5 On the **Code Generation** pane, click **Browse** to open the System Target File Browser. In the browser, select `grt.tlc` or an ERT based target file and click **OK**.
 - 6 On the **Code Generation > Interface** pane, in the **Data exchange interface** subgroup, select **ASAP2 interface**.
 - 7 Select the **Generate code only** check box on the **Code Generation** pane.
 - 8 Click **Apply**.
 - 9 Press **Ctrl+B** to build the model.

The Simulink Coder code generator writes the ASAP2 file to the build folder. By default, the file is named `model.a2l`, where `model` is the name of the model. The ASAP2 setup file controls the ASAP2 file name. For details, see “Customize Generated ASAP2 File” on page 36-2.

Use the ASAM-ASAP2 Data Definition Target

The procedure for generating the ASAP2 data definition for a model using the ASAM-ASAP2 Data Definition Target is as follows:

- 1 Create the desired model. Use parameter names and signal labels to refer to corresponding CHARACTERISTIC records and MEASUREMENT records, respectively.

- 2 Create `Simulink.Parameter` and `Simulink.Signal` objects in the MATLAB workspace and reference them from parameters and signals in the model. A convenient way of creating multiple signal and parameter data objects is to use the Data Object Wizard. Alternatively, you can create data objects one at a time from the MATLAB command line. For details on how to use the Data Object Wizard, see “Create Data Objects for a Model Using Data Object Wizard” in the Simulink documentation.
- 3 For each data object, configure the **Storage class** property to a setting other than `Auto`, `FileScope`, or `SimulinkGlobal`. This setting declares the data object as global in the generated code. For example, a storage class setting of `ExportedGlobal` configures the data object as unstructured global in the generated code.

Note: The data object is not represented in the ASAP2 file if any of the following conditions exist:

- You set the storage class to `Auto`, `FileScope`, or `SimulinkGlobal`.
- You set the storage class to `Custom` and custom storage class settings cause the code generator to generate a macro or non-addressable variable.

-
- 4 Configure the remaining properties as desired for each data object.
 - 5 On the **Code Generation** pane, click **Browse** to open the System Target File Browser. In the browser, select `asap2.tlc` and click **OK**.
 - 6 Select the **Generate code only** check box on the **Code Generation** pane.
 - 7 Click **Apply**.
 - 8 Press **Ctrl+B**.

The Simulink Coder code generator writes the ASAP2 file to the build folder. By default, the file is named `model.a2l`, where `model` is the name of the model. The ASAP2 setup file controls the ASAP2 file name. For details, see “Customize Generated ASAP2 File” on page 36-2.

Generate ASAP2 Files for Referenced Models

The build process can generate an ASAP2 file for each referenced model in a model reference hierarchy. In the generated ASAP2 file, MEASUREMENT records represent signals and states inside the referenced model.

To generate ASAP2 files for referenced models, select ASAP2 file generation for the top model and for each referenced model in the reference hierarchy. For example, if you are using the Generic Real-Time Target or an Embedded Coder target, follow the procedure described in “Use GRT or ERT Target” on page 29-9 for the top model and each referenced model.

Merge ASAP2 Files for Top and Referenced Models

Use function `rtw.asap2MergeMdlRefs` to merge the ASAP2 files generated for top and referenced models. The function has the following syntax:

```
[status,info]=rtw.asap2MergeMdlRefs(topModelName,asap2FileName)
```

- `topModelName` is the name of the model containing one or more referenced models.
- `asap2FileName` is the name you specify for the merged ASAP2 file.
- *Optional*:: `status` returns false (logical 0) if the merge completes and true (logical 1) otherwise.
- *Optional*:: `info` returns additional information about merge failure if `status` is true. Otherwise, it returns an empty character vector.

Consider the following example.

```
[status,info]=rtw.asap2MergeMdlRefs('myTopMdl','merged.a21')
```

This command merges the ASAP2 files generated for the top model `myTopMdl` and its referenced models in the file `merged.a21`.

The example model `rtwdemo_asap2` includes an example of merging ASAP2 files.

Structure of the ASAP2 File

The following table outlines the basic structure of the ASAP2 file and describes the Target Language Compiler (TLC) functions and files used to create each part of the file:

- Static parts of the ASAP2 file are shown in **bold**.
- Function calls are indicated by `%<FunctionName(>`.

File Section	Contents of <code>asap2main.tlc</code>	TLC File Containing Function Definition
File header	<code>%<ASAP2UserFcnWriteFileHead(></code>	<code>asap2userlib.tlc</code>
/begin PROJECT " "	/begin PROJECT <code>"%<ASAP2ProjectName>"</code>	<code>asap2setup.tlc</code>

File Section	Contents of asap2main.tlc	TLC File Containing Function Definition
/begin HEADER " " HEADER contents	/begin HEADER "%<ASAP2HeaderName>" %<ASAP2UserFcnWriteHeader()>	asap2setup.tlc asap2userlib.tlc
/end HEADER	/end HEADER	
/begin MODULE " " MODULE contents:	/begin MODULE "%<ASAP2ModuleName>"}	asap2setup.tlc asap2userlib.tlc
- A2ML - MOD_PAR - MOD_COMMON ...	%<ASAP2UserFcnWriteHardwareInterface()>	
Model-dependent MODULE contents:	%<SLibASAP2WriteDynamicContents()> Calls user-defined functions:	asap2lib.tlc
- RECORD_LAYOUT - CHARACTERISTIC - ParameterGroups - ModelParameters	...WriteRecordLayout_TemplateName() ...WriteCharacteristic_TemplateName() ...WriteCharacteristic_Scalar()	user/templates/...
- MEASUREMENT - ExternalInputs - BlockOutputs	...WriteMeasurement()	asap2userlib.tlc
- COMPU_METHOD	...WriteCompuMethod()	asap2userlib.tlc
/end MODULE	/end MODULE	
File footer/tail	%<ASAP2UserFcnWriteFileTail()>	asap2userlib.tlc

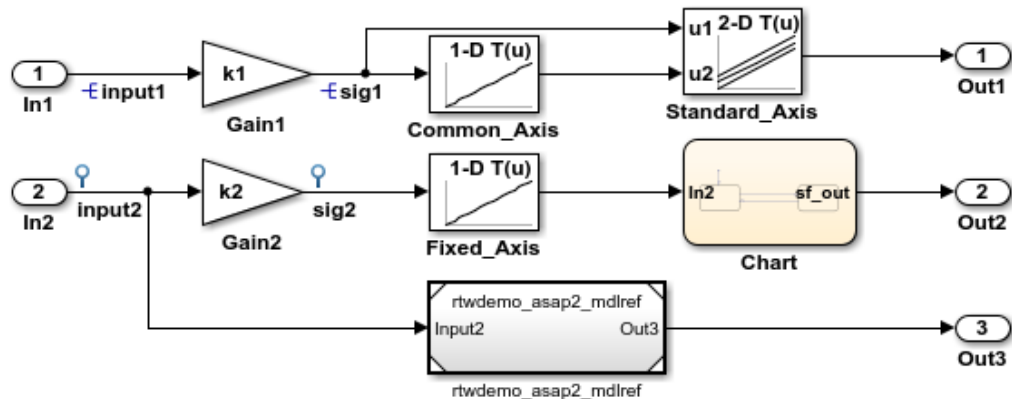
Create a Host-Based ASAM-ASAP2 Data Definition File for Data Measurement and Calibration

This model shows ASAP2 data export. ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM).

Open Example Model

Open the example model `rtwdemo_asap2`.

```
open_system('rtwdemo_asap2');
```



This model shows ASAP2 data export. ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a non-object-oriented description of the data used for measurement, calibration, and diagnostics systems. For more information on ASAM and the ASAP2 standard, see the ASAM Web site: <http://www.asam.de>.

ASAP2 data definition is achieved with Simulink data objects and test point signals. Using the Target Language Compiler (TLC), you can create highly customized solutions for your application. See the Simulink Coder documentation for details on ASAP2 file generation.

You can configure ASAP2 file generation from "Code Generation > Interface > ASAP2 interface" in the Configuration Parameters dialog box.

**Generate Code Using
Simulink Coder
(double-click)**

**Generate Code Using
Embedded Coder
(double-click)**

**View Interface
Configuration
(double-click)**

**View Workspace
Variables
(double-click)**

Copyright 1994-2014 The MathWorks, Inc.

ASAP2 is a non-object-oriented description of the data used for measurement, calibration, and diagnostics systems. For more information on ASAM and the ASAP2 standard, see the ASAM Web site: <http://www.asam.de>.

ASAP2 data definition is achieved with Simulink® data objects and test point signals. Using the Target Language Compiler (TLC), you can create highly customized solutions

for your application. See the Simulink Coder® documentation for details on ASAP2 file generation.

You can configure ASAP2 file generation by selecting **ASAP2 interface** on the **Code Generation > Interface** pane of the Configuration Parameters dialog box.

Related Examples

- “Create Tunable Calibration Parameter in the Generated Code” on page 14-56

Direct Memory Access to Generated Code for Simulink Coder

Access Memory in Generated Code Using Global Data Map

Simulink Coder provides a Target Language Compiler (TLC) function library that lets you create a *global data map record*. The global data map record, when generated, is added to the `CompiledModel` structure in the `model.rtw` file. The global data map record is a database containing information required for accessing memory in the generated code, including

- Signals (Block I/O)
- Parameters
- Data type work vectors (DWork)
- External inputs
- External outputs

Use of the global data map requires knowledge of the Target Language Compiler and of the structure of the `model.rtw` file. See “Target Language Compiler Overview” for information on these topics.

The TLC functions that are required to generate and access the global data map record are contained in `matlabroot/rtw/c/tlc/mw/globalmaplib.tlc`. The comments in the source code fully document the global data map structures and the library functions.

The global data map structures and functions might be modified or enhanced in future releases.

Performance

Optimizations for Generated Code in Simulink Coder

- “Optimization Parameters for Code Generation” on page 31-3
- “Increase Code Generation Speed” on page 31-7
- “Design Tips for Optimizing Generated Code for Stateflow Objects” on page 31-10
- “Control Compiler Optimizations” on page 31-12
- “Optimization Tools and Techniques” on page 31-13
- “Control Memory Allocation for Time Counters” on page 31-15
- “Execution Profiling for Generated Code” on page 31-16
- “Optimize Generated Code by Combining Multiple for Constructs” on page 31-19
- “Subnormal Number Performance” on page 31-22
- “Remove Code From Floating-Point to Integer Conversions That Wraps Out-of-Range Values” on page 31-27
- “Remove Code That Maps NaN to Integer Zero” on page 31-30
- “Disable Nonfinite Checks or Inlining for Math Functions” on page 31-34
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 31-40
- “Inline Invariant Signals” on page 31-43
- “Inline Numeric Values of Block Parameters” on page 31-47
- “Configure Loop Unrolling Threshold” on page 31-53
- “Use memcpy Function to Optimize Generated Code for Vector Assignments” on page 31-56
- “Generate Target Optimizations Within Algorithm Code” on page 31-60
- “Remove Code for Blocks That Have No Effect on Computational Results” on page 31-62
- “Eliminate Dead Code Paths in Generated Code” on page 31-65

- “Floating-Point Multiplication to Handle a Net Slope Correction” on page 31-68
- “Use Conditional Input Branch Execution” on page 31-71
- “Optimize Generated Code for Complex Signals” on page 31-77
- “Speed Up Linear Algebra in Code Generated from a MATLAB Function Block” on page 31-79
- “Optimize Memory Usage for Time Counters” on page 31-83
- “Minimize Memory Requirements During Code Generation” on page 31-88
- “Optimize Generated Code Using Boolean Data for Logical Signals” on page 31-89
- “Reduce Memory Usage for Boolean and State Configuration Variables” on page 31-92
- “Customize Stack Space Allocation” on page 31-93
- “Optimize Generated Code Using `memset` Function” on page 31-95
- “Vector Operation Optimization” on page 31-99
- “Enable and Reuse Local Block Outputs in Generated Code” on page 31-102

Optimization Parameters for Code Generation

In this section...

“Basic Optimization Parameters” on page 31-3

“Optimization Parameter Dependencies” on page 31-4

Many parameters, on the **All Parameters** tab and on the **Optimization and Optimization > Signals and Parameters** panes of the Configuration Parameters dialog box, depending on their setting, modify the generated code.

Basic Optimization Parameters

Some basic optimization suggestions:

- Selecting the **Signal storage reuse** check box directs the code generator to store signals in reusable memory locations.

Disabling **Signal storage reuse** makes block outputs global and unique, which in many cases significantly increases RAM and ROM usage. For more information, see “Reduce Memory Requirements for Signals”.

- Setting **Default parameter behavior** to **Inlined** reduces global RAM usage, because parameters are not declared in the global parameters structure. You can override the inlining of individual parameters by creating parameter objects. For more information, see “Default parameter behavior” and “Block Parameter Representation in the Generated Code” on page 14-43.
- Selecting the **Enable local block outputs** check box declares block signals locally in functions instead of globally (when possible). You must select **Signal storage reuse** to enable **Enable local block outputs**.
- Selecting the **Reuse local block outputs** check box reduces stack size where signals are buffered in local variables. You must select **Signal storage reuse** to enable **Reuse local block outputs**.
- Selecting the **Inline invariant signals** check box directs the code generator to not generate code for blocks with a constant (invariant) sample time. You must set **Default parameter behavior** to **Inlined** to enable **Inline invariant signals**. For more information, see “Inline Invariant Signals” on page 31-43.
- Selecting the **Eliminate superfluous local variables (expression folding)** check box minimizes the computation of intermediate results at block outputs and the storage of such results in temporary buffers or variables. For more information, see

“Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 31-40.

- Set a **Loop unrolling threshold**. The loop unrolling threshold determines when a wide signal should be wrapped into a `for` loop or when it should be generated as a separate statement for each element of the signal. For more information, see “Configure Loop Unrolling Threshold” on page 31-53.
- Specifying a **Maximum stack size (bytes)** provides control of the number of local and global variables, which determine the amount of stack space required for an application. For more information, see “Customize Stack Space Allocation” on page 31-93.
- If your target environment supports the `memcpy` function, and if your model uses signal vector assignments to move large amounts of data, selecting the **Use memcpy for vector assignment** check box can improve the execution speed of vector assignments by replacing `for` loops with `memcpy` function calls in the generated code. Set a **Memcpy threshold (bytes)**. For more information, see “Use memcpy Function to Optimize Generated Code for Vector Assignments” on page 31-56.

For more information on these parameters, see

- “Optimization Pane: General”
- “Optimization Pane: Signals and Parameters”

Optimization Parameter Dependencies

Several parameters available on the **Optimization** panes and **All Parameters** tab are dependent on another configuration parameter setting. The following tables summarize the dependencies.

Optimization > General Pane	Dependency Details
Remove root level I/O zero initialization	ERT targets only
Remove internal data zero initialization	ERT targets only

Optimization > Signals and Parameters pane	Dependency Details
Inline invariant signals	Enabled when you set Default parameter behavior to Inlined
Simplify array indexing	ERT targets only

Optimization > Signals and Parameters pane	Dependency Details
Memcpy threshold (bytes)	Enabled by Use memcpy for vector assignment
Pack Boolean data into bitfields	ERT targets only
Bitfield declarator type specifier	<ul style="list-style-type: none"> Enabled by Pack Boolean data into bitfields ERT targets only
Pass reusable subsystem output as	ERT targets only
Parameter structure	<ul style="list-style-type: none"> Enabled when you set Default parameter behavior to Inlined ERT targets only

All Parameters Tab	Dependency Details
Remove code from floating-point to integer conversions with saturation that maps NaN to zero	For ERT targets, enabled by Support floating-point numbers and Support non-finite numbers in the Code Generation > Interface pane
Remove code that protects against division arithmetic exceptions	ERT targets only
Remove reset function	ERT targets only
Remove disable function	ERT targets only
Enable local block outputs	Enabled by Signal storage reuse
Reuse local block outputs	Enabled by Signal storage reuse
Eliminate superfluous local variables (expression folding)	Enabled by Signal storage reuse

For more information on these parameters, see “Optimization Pane: General” and “Optimization Pane: Signals and Parameters”.

See Also

“Fixed Point”

Related Examples

- “Optimization Tools and Techniques” on page 31-13

- “Signal Representation in Generated Code” on page 14-109
- “Default Data Structures in the Generated Code” on page 14-16

Increase Code Generation Speed

In this section...

“Build a Model in Increments” on page 31-7

“Build Large Model Reference Hierarchies in Parallel” on page 31-7

“Minimize Memory Requirements During Code Generation” on page 31-8

“Generate Only Code” on page 31-9

“No Creation of a Code Generation Report” on page 31-9

The amount of time it takes to generate code for a model depends on the size and configuration settings of the model. For instance, if you are working with a large model, it can take awhile to generate code. To decrease the amount of time for code generation of a model, try one or more of the following methods:

- Build a model in increments
- Build large model reference hierarchies in parallel
- Minimize memory requirements during code generation
- Generate only code
- Disable the creation of a code generation report

Build a Model in Increments

You can use the `rtwbuild` command to build a model and generate code. By default, when rebuilding a model, `rtwbuild` provides an incremental model build, which only rebuilds a model or submodels that have changed since the most recent model build. Incremental model build saves code generation time. Use the **Rebuild** parameter on the **Model Referencing** pane to change the method that Simulink uses to determine when to rebuild code for referenced models. For more information on the **Rebuild** parameter, see “Rebuild”.

Build Large Model Reference Hierarchies in Parallel

In a parallel computing environment, whenever conditions allow, you can increase the speed of code generation and compilation by building models containing large model reference hierarchies in parallel. For example, if you have Parallel Computing Toolbox software, you can distribute code generation and compilation for each referenced

model across the cores of a multicore host computer. If you have MATLAB Distributed Computing Server software, you can distribute code generation and compilation for each referenced model across remote workers in your MATLAB Distributed Computing Server configuration.

The performance gain realized by using parallel builds for referenced models depends on several factors, including:

- How many models can be built in parallel for a given model referencing hierarchy
- The size of the referenced models
- Parallel computing resources such as the number of local and remote workers available
- The hardware attributes of the local and remote machines (amount of RAM, number of cores, and so on)

For more information, see “Reduce Build Time for Referenced Models” on page 25-45.

Minimize Memory Requirements During Code Generation

Models that have large amounts of parameter and constant data (such as lookup tables) can tax memory resources and slow code generation. The code generator copies this data to the *model.rtw* file. The *model.rtw* file is a partial representation of the model that the Target Language Compiler parses to transform block computations, parameters, signals, and constant data into a high-level language (for example, C). The Target Language Compiler (TLC) is an integral part of the code generator. The code generator copies parameters and data into *model.rtw*, whether they originate in the model or come from variables or objects in a workspace.

You can improve code generation speed by specifying the maximum number of elements that data vectors can have for the code generator to copy this data to *model.rtw*. When a data vector exceeds the specified size, the code generator places a reference key in *model.rtw*. The TLC uses this key to access the data from Simulink and format it into the generated code. Reference keys result in maintaining only one copy of large data vectors in memory.

The default value above which the code generator uses reference keys in place of actual data values is 10 elements. You can verify this value. In the Command Window, type the following command:

```
get_param(0, 'RTWDataReferencesMinSize')
```


To set the threshold to a different value, in the Command Window, type the following `set_param` function:

```
set_param(0, 'RTWDataReferencesMinSize', <size>)
```

Provide an integer value for `size` that specifies the number of data elements above which the code generator uses reference keys in place of actual data values.

Generate Only Code

You can increase code generation speed by specifying that the build process generate code and a makefile, but not invoke the make command. When the code generator invokes the make command, the build process takes longer because the code generator generates code, compiles code, and creates an executable file.

On the **Code Generation** pane in the Model Configuration Parameters dialog box, you can specify that the build process generate only code by selecting the **Generate code only** parameter. You can specify that the code generation process build a makefile by selecting the **Generate makefile** parameter on the **All Parameters** tab.

No Creation of a Code Generation Report

You can speed up code generation by not generating a code generation report as a part of the build process. To disable the creation of a code generation report, on the **Code Generation > Report** pane, clear the **Create code generation report** parameter. After the build process, you can generate a code generation report by doing this procedure, “Generate Code Generation Report After Build Process” on page 20-6.

Related Examples

- “Enable parallel model reference builds”
- “MATLAB worker initialization for builds”
- “Reduce Build Time for Referenced Models” on page 25-45

Design Tips for Optimizing Generated Code for Stateflow Objects

In this section...

“Do Not Access Machine-Parented Data In a Graphical Function” on page 31-10

“Be Explicit About the Inline Option of a Graphical Function” on page 31-10

“Avoid Using Multiple Edge-Triggered Events in Stateflow Charts” on page 31-10

“Combine Input Signals of a Chart Into a Single Bus Object” on page 31-10

“Use Charts with Discrete Sample Times” on page 31-11

Do Not Access Machine-Parented Data In a Graphical Function

This restriction prevents long parameter lists from appearing in the code generated for a graphical function. You can access local data that resides in the same chart as the graphical function.

For more information, see “Reuse Logic Patterns Using Graphical Functions”.

Be Explicit About the Inline Option of a Graphical Function

When you use a graphical function in a Stateflow chart, select **Inline** or **Function** for the property **Function Inline Option**. Otherwise, the code generated for a graphical function may not appear as you want.

For more information, see “Specify Graphical Function Properties”.

Avoid Using Multiple Edge-Triggered Events in Stateflow Charts

If you use more than one edge trigger, you generate multiple code statements to handle rising or falling edge detections. If multiple triggers are required, use function-call events instead.

For more information, see “How Events Work in Stateflow Charts”.

Combine Input Signals of a Chart Into a Single Bus Object

When you use a bus object, you reduce the number of parameters in the parameter list of a generated function. This guideline also applies to output signals of a chart.

For more information, see “Define Stateflow Structures”.

Use Charts with Discrete Sample Times

The code generated for discrete charts that are not inside a triggered or enabled subsystem uses integer counters to track time instead of Simulink provided time. This allows for more efficient code generation in terms of overhead and memory, as well as enabling this code for use in Software-in-the-Loop(SIL) and Processor-in-the-Loop(PIL) simulation modes.

See Also

“Optimization Pane: Stateflow”

Related Examples

- “Code Generation of Stateflow Blocks” on page 13-2
- “Reduce Memory Usage for Boolean and State Configuration Variables” on page 31-92

Control Compiler Optimizations

To control compiler optimizations for a makefile build at the GUI level, use the **Compiler optimization level** parameter. The **Compiler optimization level** parameter provides

- Target-independent values **Optimizations on (faster runs)** and **Optimizations off (faster builds)**, which allow you to easily toggle compiler optimizations on and off during code development
- The value **Custom** for entering custom compiler optimization flags at the Simulink GUI level, rather than editing compiler flags into template makefiles (TMFs) or supplying compiler flags to build process make commands

The default setting is **Optimizations off (faster builds)**. Selecting the value **Custom** enables the **Custom compiler optimization flags** field, in which you can enter custom compiler optimization flags (for example, `-O2`).

Note: If you specify compiler options for your makefile build using `OPT_OPTS`, `MEX_OPTS` (except `MEX_OPTS= " -v "`), or `MEX_OPT_FILE`, the value of **Compiler optimization level** is ignored and a warning is issued about the ignored parameter.

For more information about the **Compiler optimization level** parameter and its values, see “Compiler optimization level” and “Custom compiler optimization flags”.

Related Examples

- “Template Makefiles and Make Options” on page 17-26
- “Select a System Target File” on page 17-2
- “Support Compiler Optimization Level Control” on page 37-95

Optimization Tools and Techniques

Using the Model Advisor, you can analyze a model for code generation and identify aspects of your model that impede production deployment or limit code efficiency. You can select from a set of checks to run on a model's current configuration. The Model Advisor analyzes the model and generates check results providing suggestions for improvements in each area. Most Model Advisor diagnostics do not require the model to be in a compiled state; those that do are noted.

Before running the Model Advisor, select the target you plan to use for code generation. The Model Advisor works most effectively with ERT and ERT-based system target files.

Use the following examples to investigate optimizing models for code generation using the Model Advisor:

- `rtwdemo_advisor1`
- `rtwdemo_advisor2`
- `rtwdemo_advisor3`

Note: Example models `rtwdemo_advisor2` and `rtwdemo_advisor3` require Stateflow and Fixed-Point Designer software.

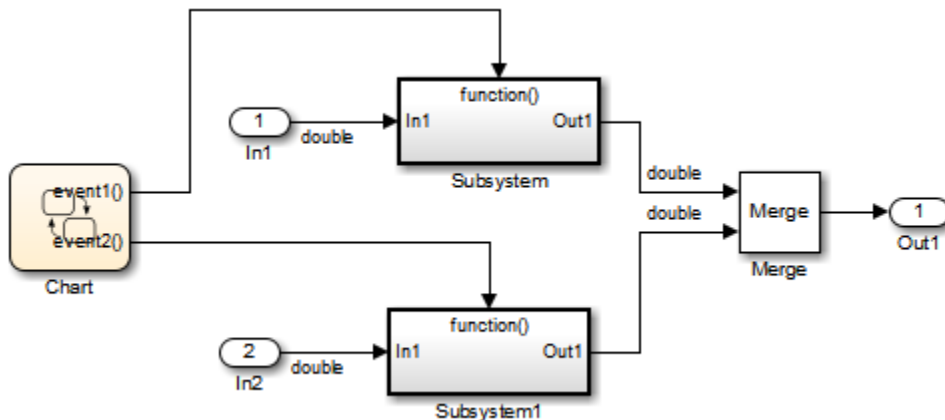
For more information on using the Model Advisor, see “Run Model Checks” in the Simulink documentation. For more information about the checks, see “Simulink Coder Checks” in the Simulink Coder documentation.

In addition to analyzing models with Model Advisor, you can also do the following:

- Use the Upgrade Advisor to upgrade older models (saved by prior versions or the current version) to use current features. For details, see “Model Upgrades”.
- Before building, set optimization flags for the compiler (for example, `-O2` for `gcc`, `-Ot` for the Microsoft Visual C++ compiler).
- Directly inline C/C++ S-functions into the generated code by writing a TLC file for the S-function. For more information, see “Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target” on page 22-33 and the Target Language Compiler documentation.
- Use a Simulink data type other than `double` when possible. The available data types are `Boolean`, signed and unsigned 8-, 16-, and 32-bit integers, and 32- and 64-bit

floats (a `double` is a 64-bit float). For more information, see “About Data Types in Simulink”. For a block-by-block summary, click `showblockdatatypetable` or type the command in the Command Window.

- For tunable block parameters that you configure to store in memory in the generated code, you can match parameter data types with signal data types to eliminate unnecessary typecasts and C shifts. Where possible, store parameter values in small integer data types. See “Parameter Data Types in the Generated Code” on page 14-76.
- Remove repeated values in lookup table data.
- Use the Merge block to merge the output of signals wherever possible. This block is particularly helpful when you need to control the execution of function-call subsystems with a Stateflow chart. The following model shows an example of how to use the Merge block.



When more than one signal connected to a Merge block has a non-`Auto` storage class, all non-`Auto` signals connected to that block must *be identically labeled* and *have the same storage class*. When Merge blocks connect directly to one another, these rules apply to the signals connected to any of the Merge blocks in the group.

Related Examples

- “Optimization Parameters for Code Generation” on page 31-3
- “Increase Code Generation Speed” on page 31-7
- “Execution Profiling for Generated Code” on page 31-16

Control Memory Allocation for Time Counters

The **Application lifespan (days)** parameter lets you control the allocation of memory for absolute and elapsed time counters. Such counters exist in the code for blocks that use absolute or elapsed time. For a list of such blocks, see “Absolute Time Limitations” on page 3-12.

The size of the time counters in generated code is 8, 16, 32, or 64 bits. The size is set automatically to the minimum that can accommodate the duration value specified by **Application lifespan (days)** given the step size specified in the Configuration Parameters **Solver** pane. To minimize the amount of RAM used by time counters, specify the smallest lifespan possible and the largest step size possible.

An application runs to its specified lifespan. It may be able to run longer. For example, running a model with a step size of one millisecond (0.001 seconds) for one day requires a 32-bit timer, which could continue running without overflow for 49 days more.

To maximize application lifespan, specify **Application lifespan (days)** as `inf`. This value allocates 64 bits (two `uint32` words) for each timer. Using 64 bits to store timing data would allow a model with a step size of 0.001 microsecond (10E-09 seconds) to run for more than 500 years, which would rarely be required. 64-bit counters do not violate the usual code generator length limitation of 32 bits because the value of a time counter does not provide the value of a signal, state, or parameter.

See Also

“Application lifespan (days)”

Related Examples

- “Absolute and Elapsed Time Computation” on page 3-2
- “Timers in Asynchronous Tasks” on page 5-42

Execution Profiling for Generated Code

Use code execution profiling to:

- Determine whether the generated code meets execution time requirements for real-time deployment on your target hardware.
- Identify code sections that require performance improvements.

The following tasks represent a general workflow that uses code execution profiling:

- 1** With the Simulink model, design and optimize your algorithm.
- 2** Configure the model for code execution profiling, and generate code.
- 3** Execute generated code on target. For example, you can:
 - Run a software-in-the-loop (SIL) simulation on your development computer.
 - Run a processor-in-the-loop (PIL) simulation using a target support package or custom PIL target.
 - Perform real-time execution with Simulink Real-Time or a target support package.
- 4** Analyze performance through code execution profiling plots and reports. For example, check that the algorithm code satisfies execution time requirements for real-time deployment:
 - If the algorithm code easily meets the requirements, consider enhancing your algorithm to exploit available processing power.
 - If the code cannot be executed in real time, look for ways to reduce execution time.

Identify the tasks that require the most time. For these tasks, investigate whether trade-offs between functionality and speed are possible.

If your target is a multicore processor, distribute the execution of algorithm code across available cores.

- 5** If required, refine the model and return to step 2.

To find information about code execution profiling with Simulink products, use the following table.

Target	Execution Feature	Type of Profiling	Relevant Products	See
Development computer	Model configured for concurrent execution	Execution time	Simulink Coder	<ul style="list-style-type: none"> • “Optimize and Deploy on a Multicore Target” • “Concurrent Execution Models”
Development computer	Software-in-the-loop (SIL)	Execution time	Embedded Coder	<ul style="list-style-type: none"> • “Code Execution Profiling with SIL and PIL” • “View and Compare Code Execution Times” • “Analyze Code Execution Data”
Embedded hardware or instruction set simulator	Processor-in-the-loop (PIL)	Execution time	Embedded Coder	<ul style="list-style-type: none"> • “Code Execution Profiling with SIL and PIL” • “View and Compare Code Execution Times” • “Analyze Code Execution Data”
Target support packages	Real-time execution, PIL	Execution time	Embedded Coder	<ul style="list-style-type: none"> • “Code Execution Profiling for IDE and Toolchain Targets” • “Perform Execution-Time Profiling for IDE and Toolchain Targets”
Target support packages	Real-time execution	Stack	Embedded Coder	<ul style="list-style-type: none"> • “Code Execution Profiling for IDE and Toolchain Targets” • “Perform Stack Profiling with IDE and Toolchain Targets”
Simulink Real-Time	Real-time execution	Execution time	Simulink Coder, Simulink Real-Time	<ul style="list-style-type: none"> • “Execution Profiling for Real-Time Applications” • “Configure Real-Time Application for Profiling” • “Generate Real-Time Application Execution Profile”

Target	Execution Feature	Type of Profiling	Relevant Products	See
Simulink Real-Time	Real-time execution, model configured for concurrent execution	Execution time	Simulink Coder, Simulink Real-Time	<ul style="list-style-type: none">• “Execution Profiling for Real-Time Applications”• “Concurrent Execution on Simulink® Real-Time™”

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “SIL and PIL Simulations”

Optimize Generated Code by Combining Multiple for Constructs

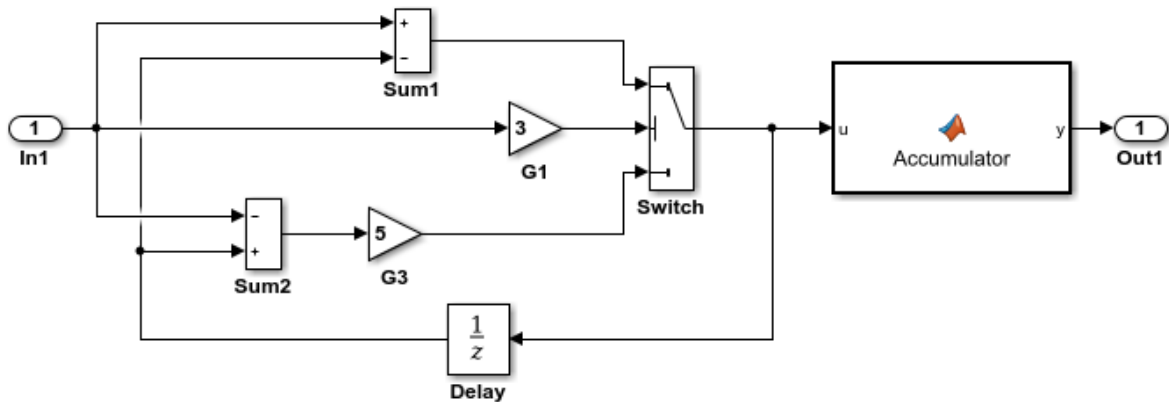
This example shows how the code generator combines for loops. The generated code uses for constructs to represent a variety of modeling patterns, such as a matrix signal or Iterator blocks. Using data dependency analysis, the code generator combines for constructs to reduce static code size and runtime branching.

The benefits of optimizing for loops are:

- Reducing ROM and RAM consumption.
- Increasing execution speed.

for Loop Modeling Patterns

In the model, `rtwdemo_forloop`, the Switch block and MATLAB Function block represent for constructs. In the In1 Block Parameters dialog box, the **Port dimensions** parameter is set to 10.



This model shows how Simulink Coder optimizes for-loops by combining multiple blocks into the same for-loop, improving code efficiency and readability. The for-loop fusion optimization complements expression folding. To see the vector widths in this model, select Edit > Update Diagram.

Generate Code Using
Simulink Coder
(double-click)

Generate Code Using
Embedded Coder
(double-click)

Copyright 1994-2012 The MathWorks, Inc.

Generate Code

In the model, there are no data dependencies across the `for` loop iterations. Therefore, the code generator combines all `for` loops into one loop. Build the model and view the generated code.

```
### Starting build procedure for model: rtwdemo_forloop
### Successful completion of build procedure for model: rtwdemo_forloop
```

The generated file, `rtwdemo_forloop.c`, contains the code for the single `for` loop.

```
/* Model step function */
void rtwdemo_forloop_step(void)
{
    int32_T k;

    /* MATLAB Function: '<Root>/Accum' */
    /* MATLAB Function 'Accum': '<S1>:1' */
    /* '<S1>:1:3' */
    /* '<S1>:1:4' */
    rtwdemo_forloop_Y.Out1 = 0.0;

    /* '<S1>:1:5' */
    for (k = 0; k < 10; k++) {
        /* Switch: '<Root>/Switch' incorporates:
         * Gain: '<Root>/G1'
         * Gain: '<Root>/G3'
         * Inport: '<Root>/In1'
         * Sum: '<Root>/Sum1'
         * Sum: '<Root>/Sum2'
         * UnitDelay: '<Root>/Delay'
         */
        if (3.0 * rtwdemo_forloop_U.In1[k] >= 0.0) {
            rtwdemo_forloop_DW.Delay_DSTATE[k] = rtwdemo_forloop_U.In1[k] -
                rtwdemo_forloop_DW.Delay_DSTATE[k];
        } else {
            rtwdemo_forloop_DW.Delay_DSTATE[k] = (rtwdemo_forloop_DW.Delay_DSTATE[k] -
                rtwdemo_forloop_U.In1[k]) * 5.0;
        }

        /* End of Switch: '<Root>/Switch' */

        /* MATLAB Function: '<Root>/Accum' */
        /* '<S1>:1:5' */
    }
}
```

```
/* '<S1>:1:6' */
rtwdemo_forloop_Y.Out1 += (1.0 + (real_T)k) +
    rtwdemo_forloop_DW.Delay_DSTATE[k];
}
}
```

Close the model.

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Configure Loop Unrolling Threshold” on page 31-53
- “For Loop”

Subnormal Number Performance

Subnormal numbers, formerly known as denormal numbers in floating-point literature, fill the underflow gap around zero in floating-point arithmetic. Subnormal values are a special category of floating-point values that are too close to 0.0 to be represented as a normalized value. The leading significand (mantissa) of a subnormal number is zero. When adding and subtracting floating-point numbers, subnormal numbers prevent underflow.

Using subnormal numbers provides precision beyond the normal representation by using leading zeros in the significand to represent smaller values after the representation reaches the minimum exponent. As the value approaches 0.0, you trade off precision for extended range. Subnormal numbers are useful if your application requires extra range.

However, in a real-time system, using subnormal numbers can dramatically increase execution latency, resulting in excessive design margins and real-time overruns. If the simulation or generated code performs calculations that produce or consume subnormal numbers, the execution of these calculations can be up to 50 times slower than similar calculations on normal numbers. The actual simulation or code execution time for subnormal number calculations depends on your computer operating environment. Typically, for desktop processors, the execution time for subnormal number calculations is five times slower than similar calculations on normal numbers.

To minimize the possibility of execution slowdowns or overruns due to subnormal number calculation latency, do one of the following:

- In your model, manually flush to zero any incoming or computed subnormal values at inputs and key operations, such as washouts and filters. For an example, see “Flush Subnormal Numbers to Zero” on page 31-24.

To detect a subnormal value for a single precision, 32-bit floating-point number:

- 1 Find the smallest normalized number on a MATLAB host. In the Command Window, type:

```
>> SmallestNormalSingle = realmin('single')  
In the C language, FLT_MIN, defined in float.h, is equivalent to  
realmin('single').
```

- 2 Look for values in range:

```
0 < fabsf(x) < SmallestNormalSingle
```

To detect a subnormal value for a double precision, 64-bit floating-point number:

- 1 Find the smallest normalized number on a MATLAB host. In the Command Window, type:

```
>> SmallestNormalDouble = realmin('double')
```

In the C language, `DBL_MIN`, defined in `float.h`, is equivalent to `realmin('double')`.

- 2 To detect a subnormal value, look for values in this range:

$$0 < \text{fabs}(x) < \text{SmallestNormalDouble}$$

- On your processor, set flush-to-zero mode or, with your compiler, specify an option to disable subnormal numbers. Flush-to-zero modes treat a subnormal number as 0 when it is an input to a floating-point operation. Underflow exceptions do not occur in flush-to-zero mode.

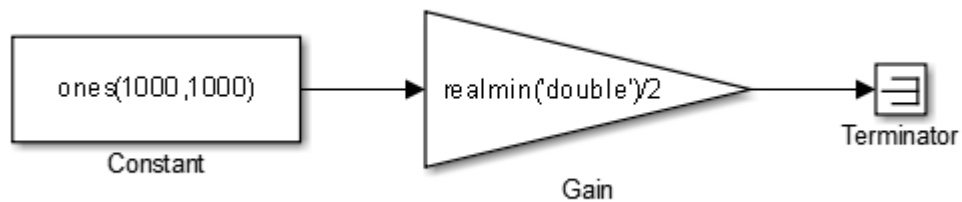
For example, in Intel® processors, the flush-to-zero (FTZ) and denormals-are-zero (DAZ) flags in the MXCSR register control floating-point calculations. For the gcc compiler on Linux, `-ffast-math` sets abrupt underflow (FTZ), flush-to-zero, while `-O3 -ffast-math` reverts to gradual underflow, using subnormal numbers.

For more information, see the IEEE Standard 754, *IEEE Standard for Floating-Point Arithmetic*.

Simulation Time With and Without Subnormal Numbers

This model shows how using subnormal numbers increases simulation time by ~5 times.

- 1 Open the model `ex_subnormal`. The Gain is set to subnormal value `realmin('double')/2`.



- 2 To run a simulation, in the Command Window, type `for k=1:5, tic; sim('ex_subnormal'); toc,end`. Observe the elapsed times for simulation using subnormals, similar to the following:

```
>> for k=1:5, tic; sim('ex_subnormal'); toc,end
Elapsed time is 9.909326 seconds.
Elapsed time is 9.617966 seconds.
Elapsed time is 9.797183 seconds.
Elapsed time is 9.702397 seconds.
Elapsed time is 9.893946 seconds.
```

- 3 Set the Gain to a number, 2, that is not a subnormal value:

```
>> set_param('ex_subnormal/Gain', 'Gain', '2');
```

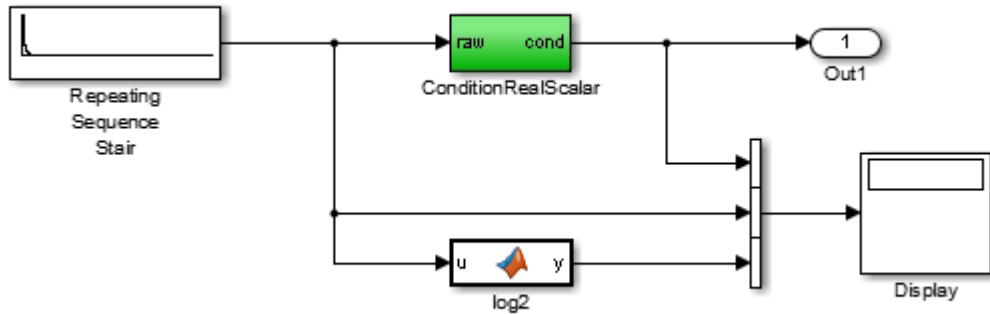
- 4 To run a simulation, in the Command Window, type `for k=1:5, tic; sim('ex_subnormal'); toc,end`. Observe elapsed times for simulations that do not use subnormal values, similar to the following:

```
>> for k=1:5, tic; sim('ex_subnormal'); toc,end
Elapsed time is 2.045123 seconds.
Elapsed time is 1.796598 seconds.
Elapsed time is 1.758458 seconds.
Elapsed time is 1.721721 seconds.
Elapsed time is 1.780569 seconds.
```

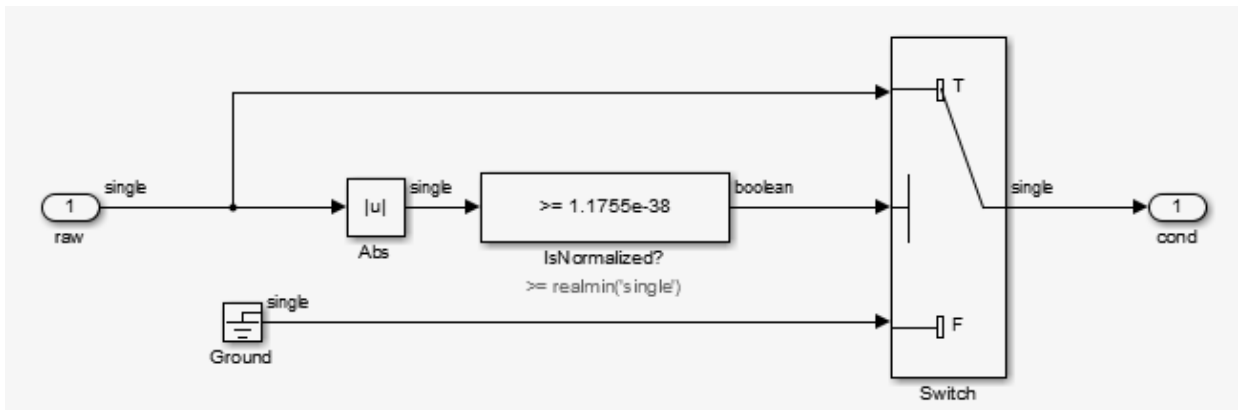
Flush Subnormal Numbers to Zero

This example shows how to flush single precision subnormal numbers to zero.

- 1 Open the model `ex_flush_to_zero`:

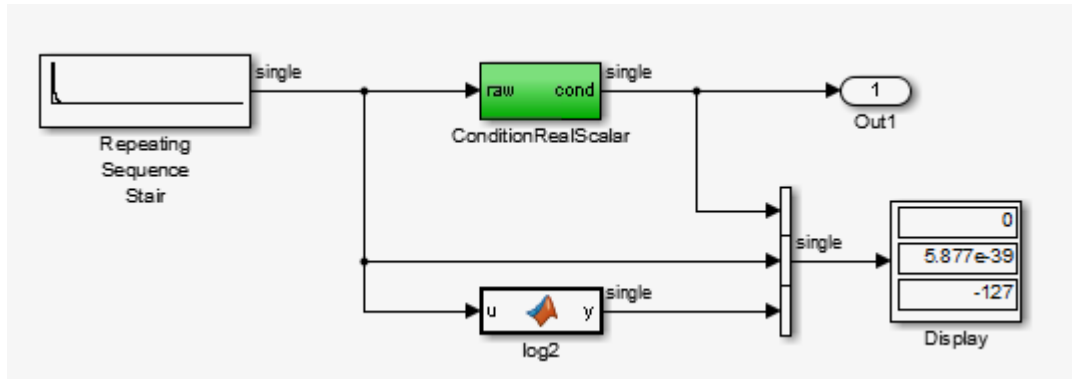


- Repeating Sequence Stair generates a sequence of numbers from two raised to the power of 0 through two raised to the power of -165. The sequence approaches zero.
- ConditionRealScalar flushes subnormal single precision values that are less than `realmin('single')` to zero.



- MATLAB function block `log2` generates the base 2 logarithm of the Repeating Sequence Stair output. Specifically, `log2` generates the numbers 0 through -165.
- 2** On the **Simulation > Stepping Options** pane:
- Select **Enable stepping back**.

- Select **Pause simulation when time reaches** and enter 121.
- 3 In the model window, run the simulation. The simulation pauses at T=121. The displayed values:
- ConditionRealScalar output approaches zero.
 - Repeating Sequence Stair output approaches zero.
- 4 Step the simulation forward to T=127. ConditionRealScalar flushes the subnormal value output from Repeating Sequence Stair to zero.



- 5 Continue stepping the simulation forward. ConditionRealScalar flushes the subnormal single precision values output from Repeating Sequence Stair to zero. When T=150, the output of Repeating Sequence Stair is itself zero.

Related Examples

- “Data Types Supported by Simulink”
- “Numerical Consistency of Model and Generated Code Simulation Results” on page 33-2

Remove Code From Floating-Point to Integer Conversions That Wraps Out-of-Range Values

In this section...

“Example Model” on page 31-27

“Generate Code Without Optimization” on page 31-28

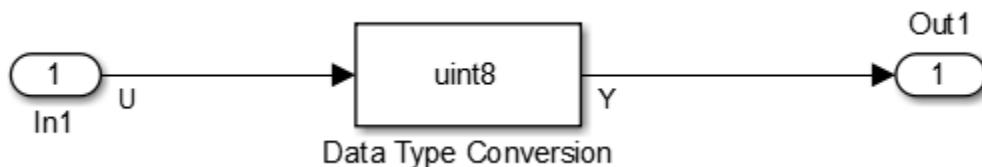
“Generate Code with Optimization” on page 31-29

This example shows how to remove code for out-of-range floating-point to integer conversions. Without this code, there might be a mismatch between simulation and code generation results. Standard C does not define the behavior of out-of-range floating-point to integer conversions, while these conversions are well-defined during simulation. In Standard C and during simulation, floating-point to integer conversions are well-defined for input values in the range of the output type.

If the input values in your application are in the range of the output type, remove code for out-of-range floating-point to integer conversions. Removing this code reduces the size and increases the speed of the generated code.

Example Model

In this model, a Data Type Conversion block converts an input signal from a `double` to a `uint8`. A `uint8` can support values from 0 to 255. If the input signal has a value outside of this range, an out-of-range conversion occurs. In this example, the model is named `conversion_ex`.



- 1 Use Inport, Output, and Data Type Conversion blocks to create the example model.

- 2 Open the Inport Block Parameters dialog box and select the **Signal Attributes** tab. For the **Data Type** parameter, select **double**.
- 3 Open the Data Type Conversion dialog box. For the **Output data type** parameter, select **uint8**.
- 4 For the signal feeding into the Data Type Conversion block, open the Signal Properties dialog box. Enter the name **U**. On the **Code Generation** tab, for the **Storage Class** parameter, select **ImportedExtern**.
- 5 For the signal leaving the Data Type Conversion block, open the Signal Properties dialog box. Enter the name **Y**. On the **Code Generation** tab, for the **Storage Class** parameter, select **ImportedExtern**.

Generate Code Without Optimization

- 1 Open the Model Configuration Parameters dialog box. On the **Solver** pane, for the **Type** parameter, select **Fixed-step**.
- 2 On the **Code Generation > Report** pane, select **Create code generation report**.
- 3 On the **Code Generation** pane, select **Generate code only**, and then, in the model window, press **Ctrl+B**. When code generation is complete, an HTML code generation report opens.
- 4 In the Code Generation report, select the `conversion_ex.c` file and view the model step function. The code generator applies the `fmod` function to handle out-of-range results.

```
/* Model step function */
void conversion_ex_step(void)
{
    real_T tmp;

    /* DataTypeConversion: '<Root>/Data Type Conversion' incorporates:
     * Inport: '<>/In1'
     */
    tmp = floor(U);
    if (rtIsNaN(tmp) || rtIsInf(tmp)) {
        tmp = 0.0;
    } else {
        tmp = fmod(tmp, 256.0);
    }

    Y = (uint8_T)(tmp < 0.0 ? (int32_T)(uint8_T)-(int8_T)(uint8_T)-tmp : (int32_T)
        (uint8_T)tmp);
}
```

Generate Code with Optimization

- 1 Open the Configuration Parameters dialog box. On the **Optimization** pane, select **Remove code from floating-point to integer conversions that wraps out-of-range values**. Generate code.
- 2 In the code generation report, select the `conversion_ex.c` file and view the model step function. The generated code does not contain code that protects against out-of-range values.

```
/* Model step function */
void conversion_ex_step(void)
{
    /* DataTypeConversion: '<Root>/Data Type Conversion' incorporates:
     * Inport: '<Root>/In1'
     */
    Y = (uint8_T)U;
}
```

The generated code is more efficient without this protective code, but it is possible that the execution of generated code does not produce the same results as simulation for values not in the range of 0 to 255.

See Also

“Remove code from floating-point to integer conversions that wraps out-of-range values”

Related Examples

- “his1_0053: Configuration Parameters > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values”
- “Optimization Tools and Techniques” on page 31-13
- “Remove Code That Maps NaN to Integer Zero” on page 31-30

Remove Code That Maps NaN to Integer Zero

In this section...

“Example Model” on page 31-30

“Generate Code” on page 31-31

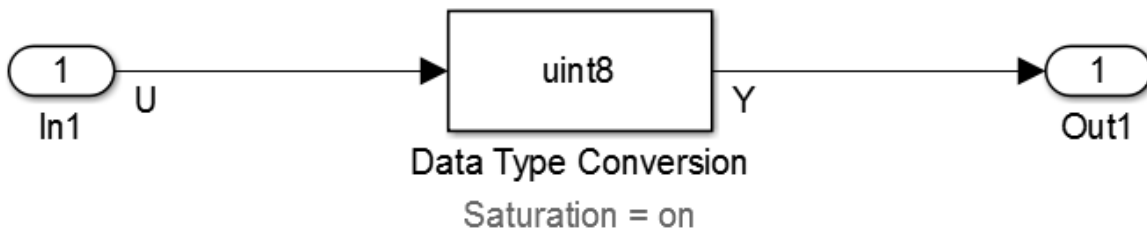
“Generate Code with Optimization” on page 31-32

This example shows how to remove code that maps NaN to integer zero. For floating-point to integer conversions involving saturation, Simulink converts NaN to integer zero during simulation. If your model contains an input value of NaN, you can specify that the code generator produce code that maps NaN to zero. Without this code, there is a mismatch between simulation and code generation results because in Standard C, every condition involving NaN evaluates to false.

If there are no input values of NaN in your application, you can remove code that maps NaN to integer zero. Removing this code reduces the size and increases the speed of the generated code.

Example Model

In this model, a Data Type Conversion block converts an input signal from a double to a uint8. In this example, the model is named `conversion_ex`.



- 1 Use Inport, Outport, and Data Type Conversion blocks to create the example model.
- 2 Open the Inport Block Parameters dialog box and click the **Signal Attributes** tab. For the **Data Type** parameter, select `double`.
- 3 Open the Data Type Conversion dialog box. For the **Output data type** parameter, select `uint8`.

- 4 Select **Saturate on integer overflow**. Selecting this parameter specifies that an out-of-range signal value equals either the minimum or maximum value that the data type can represent.
- 5 For the signal feeding into the Data Type Conversion block, open the Signal Properties dialog box. Enter a name of U. On the **Code Generation** tab, for the **Storage Class** parameter, select **ImportedExtern**.
- 6 For the signal leaving the Data Type Conversion block, open the Signal Properties dialog box. Enter a name of Y. On the **Code Generation** tab, for the **Storage Class** parameter, select **ImportedExtern**.

Generate Code

- 1 Open the Model Configuration Parameters dialog box. On the **Solver** pane, for the **Type** parameter, select **Fixed-step**.
- 2 Open the Model Configuration Parameters dialog box. On the **All Parameters** tab, clear **Remove code from floating-point to integer conversions with saturation that maps NaN to zero**.
- 3 On the **Code Generation > Report** pane, select **Create code generation report**.
- 4 On the **Code Generation** pane, select **Generate code only** and then, in the model window, press **Ctrl+B**. When code generation is complete, an HTML code generation report opens.
- 5 In the Code Generation report, select the `nan_int_ex.c` file and view the model step function. For an input value of NaN, there is agreement between the generated code and simulation because NaN maps to integer zero.

```

/* Model step function */
void nan_int_ex_step(void)
{
    /* DataTypeConversion: '<Root>/Data Type Conversion' incorporates:
     * Inport: '<Root>/In1'
     */
    if (U < 256.0) {
        if (U >= 0.0) {
            Y = (uint8_T)U;
        } else {
            Y = 0U;
        }
    } else if (U >= 256.0) {

```

```
    Y = MAX_uint8_T;
} else {
    Y = 0U;
}
```

Generate Code with Optimization

- 1 Open the Configuration Parameters dialog box. On the **All Parameters** tab, select **Remove code from floating-point to integer conversions that wraps out-of-range values**. Generate code.
- 2 In the Code Generation report, select the `nan_int_ex.c` section and view the model step function. The generated code maps NaN to 255 and not integer zero. The generated code is more efficient without the extra code that maps NaN to integer zero, but the execution of the generated code does not produce the same results as simulation for NaN values.

```
/* Model step function */
void nan_int_ex_step(void)
{
    /* DataTypeConversion: '<Root>/Data Type Conversion' incorporates:
    * Inport: '<Root>/In1'
    */
    if (U < 256.0) {
        if (U >= 0.0) {
            Y = (uint8_T)U;
        } else {
            Y = 0U;
        }
    } else {
        Y = MAX_uint8_T;
    }

    /* End of DataTypeConversion: '<Root>/Data Type Conversion' */
}
```

See Also

“Remove code from floating-point to integer conversions with saturation that maps NaN to zero”

Related Examples

- “Optimization Tools and Techniques” on page 31-13

- “Remove Code From Floating-Point to Integer Conversions That Wraps Out-of-Range Values” on page 31-27

Disable Nonfinite Checks or Inlining for Math Functions

When the code generator produces code for math functions:

- If the model option **Support non-finite numbers** is selected, nonfinite number checking is generated uniformly for math functions, without the ability to specify that nonfinite number checking should be generated for some functions, but not for others.
- By default, inlining is applied uniformly for math functions, without the ability to specify that inlining should be generated for some functions, while invocations should be generated for others.

You can use code replacement library (CRL) customization entries to:

- Selectively disable nonfinite checks for math functions. This can improve the execution speed of the generated code.
- Selectively disable inlining of math functions. This can increase code readability and decrease code size.

The functions for which these customizations are supported include the following:

- Floating-point only: `atan2`, `copysign`, `fix`, `hypot`, `log`, `log10`, `round`, `sincos`, and `sqrt`
- Floating-point and integer: `abs`, `max`, `min`, `mod`, `rem`, `saturate`, and `sign`

The general workflow for disabling nonfinite number checking and/or inlining is as follows:

- 1 If you can disable nonfinite number checking for a particular math function, or if you want to disable inlining for a particular math function and instead generate a function invocation, you can copy the following MATLAB function code into a MATLAB file with an `.m` file name extension, for example, `crl_table_customization.m`.

```
function hTable = crl_table_customization

% Create an instance of the Code Replacement Library table for controlling
% function intrinsic inlining and nonfinite support

hTable = RTW.Tf1Table;

% Inline - true (if function needs to be inline)
%           false (if function should not be inlined)
% SNF (support nonfinite) - ENABLE (if non-finite checking should be performed)
%                           DISABLE (if non-finite checking should NOT be performed)
%                           UNSPECIFIED (Default behavior)
```

```

% registerCustomizationEntry(hTable, ...
%     Priority, numInputs, key, inType, outType, Inline, SNF);

registerCustomizationEntry(hTable, ...
    100, 2, 'atan2', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'atan2', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 1, 'sincos', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'sincos', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 1, 'abs', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'abs', 'single', 'single', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 1, 'abs', 'int32', 'int32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'abs', 'int16', 'int16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'abs', 'int8', 'int8', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'abs', 'integer', 'integer', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'abs', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'abs', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'abs', 'uint8', 'uint8', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 2, 'hypot', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'hypot', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 1, 'log', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'log', 'single', 'double', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 1, 'log10', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'log10', 'single', 'double', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 2, 'min', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'min', 'single', 'single', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'min', 'int32', 'int32', true, 'UNSPECIFIED');

```

```

registerCustomizationEntry(hTable, ...
    100, 2, 'min', 'int16', 'int16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'min', 'int8', 'int8', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'min', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'min', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'min', 'uint8', 'uint8', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'min', 'integer', 'integer', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 2, 'max', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'max', 'single', 'single', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'max', 'int32', 'int32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'max', 'int16', 'int16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'max', 'int8', 'int8', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'max', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'max', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'max', 'uint8', 'uint8', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'max', 'integer', 'integer', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 2, 'mod', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'mod', 'single', 'single', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'mod', 'int32', 'int32', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'mod', 'int16', 'int16', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'mod', 'int8', 'int8', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'mod', 'uint32', 'uint32', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'mod', 'uint16', 'uint16', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'mod', 'uint8', 'uint8', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 2, 'rem', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'rem', 'single', 'single', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'rem', 'int32', 'int32', false, 'UNSPECIFIED');

```

```

registerCustomizationEntry(hTable, ...
    100, 2, 'rem', 'int16', 'int16', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'rem', 'int8', 'int8', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'rem', 'uint32', 'uint32', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'rem', 'uint16', 'uint16', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 2, 'rem', 'uint8', 'uint8', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 1, 'round', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'round', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 3, 'saturate', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 3, 'saturate', 'single', 'single', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 3, 'saturate', 'int32', 'int32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 3, 'saturate', 'int16', 'int16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 3, 'saturate', 'int8', 'int8', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 3, 'saturate', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 3, 'saturate', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 3, 'saturate', 'uint8', 'uint8', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 3, 'saturate', 'integer', 'integer', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 1, 'sign', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'sign', 'single', 'single', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'sign', 'int32', 'integer', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'sign', 'int16', 'integer', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'sign', 'int8', 'integer', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'sign', 'uint32', 'uint32', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'sign', 'uint16', 'uint16', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'sign', 'uint8', 'uint8', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'sign', 'integer', 'integer', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...

```

```

    100, 1, 'sqrt', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'sqrt', 'single', 'single', true, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 1, 'fix', 'double', 'double', false, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'fix', 'single', 'single', false, 'UNSPECIFIED');

registerCustomizationEntry(hTable, ...
    100, 1, 'copysign', 'double', 'double', true, 'UNSPECIFIED');
registerCustomizationEntry(hTable, ...
    100, 1, 'copysign', 'single', 'single', true, 'UNSPECIFIED');

```

- 2 To reduce the size of the file, you can delete the `registerCustomizationEntry` lines for functions for which the default nonfinite number checking and inlining behavior is acceptable.
- 3 For each remaining entry,
 - Set the `Inline` argument to `true` if the function should be inlined or `false` if it should not be inlined.
 - Set the `SNF` argument to `ENABLE` if nonfinite checking should be generated, `DISABLE` if nonfinite checking should not be generated, or `UNSPECIFIED` to accept the default behavior based on the model option settings.

Save the file.

- 4 Optionally, perform a quick check of the syntactic validity of the customization table entries by invoking the table definition file at the MATLAB command line (`>> tbl = crl_table_customization`). Fix syntax errors that are flagged.
- 5 Optionally, view the customization table entries in the Code Replacement Viewer (`>> crviewer(crl_table_customization)`). For more information about viewing code replacement tables, see “Choose a Code Replacement Library” on page 21-8.
- 6 To register these changes and make them appear in the **Code replacement library** drop-down list located on the **Code Generation > Interface** pane of the Configuration Parameters dialog box, first copy the following MATLAB function code into an instance of the file `rtwTargetInfo.m`.

Note: For the example below, specify the argument `'RTW'` if a GRT target is selected for your model, otherwise omit the argument.

```

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

```

```

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

end % End of RTWTARGETINFO

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

% Instantiate a CRL registry entry - specify 'RTW' for GRT
thisCrl = RTW.TflRegistry('RTW');

% Define the CRL properties
thisCrl.Name = 'CRL Customization Example';
thisCrl.Description = 'Example of CRL Customization';
thisCrl.TableList = {'crl_table_customization'};
thisCrl.TargetHWDeviceType = {'*'};

end % End of LOCCRLREGFCN

```

You can edit the **Name** field to specify the library name that appears in the **Code replacement library** drop-down list. Also, the file name in the **TableList** field must match the name of the file you created in step 1.

To register your changes, with both of the MATLAB files you created present in the MATLAB path, enter the following command at the MATLAB command line:

```
sl_refresh_customizations
```

- 7 Create or open a model that generates function code corresponding to one of the math functions for which you specified a change in nonfinite number checking or inlining behavior.
- 8 Open the Configuration Parameters dialog box, go to the **Code Generation** > **Interface** pane, and use the **Code replacement library** drop-down list to select the code replacement entry you registered in step 6, for example, **CRL Customization Example**.
- 9 Generate code for the model and examine the generated code to verify that the math functions are generated as expected.

Minimize Computations and Storage for Intermediate Results at Block Outputs

In this section...

“Expression Folding” on page 31-40

“Example Model” on page 31-40

“Generate Code” on page 31-41

“Enable Optimization” on page 31-41

“Generate Code with Optimization” on page 31-42

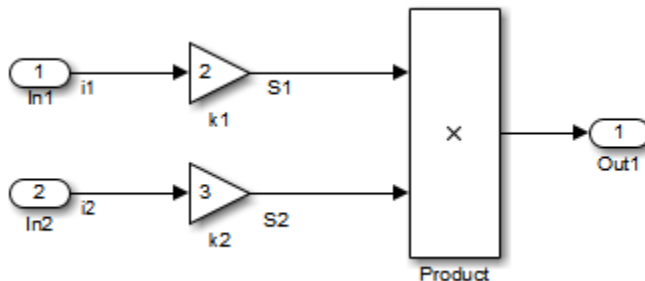
Expression Folding

Expression folding optimizes code to minimize the computation of intermediate results at block outputs and the storage of such results in temporary buffers or variables. When expression folding is on, the code generator collapses (folds) block computations into a single expression, instead of generating separate code statements and storage declarations for each block in the model. Most Simulink blocks support expression folding.

Expression folding improves the efficiency of generated code, frequently achieving results that compare favorably to hand-optimized code. In many cases, entire groups of model computations fold into a single, highly optimized line of code.

You can use expression folding in your own inlined S-function blocks. For more information, see “S-Functions That Support Expression Folding” on page 24-135.

Example Model



Generate Code

With expression folding off, in the `explfld.c` file, the code generator generates this code.

```
/* Model step function */
void exprfld_step(void)
{
    /* Gain: '<Root>/Gain' incorporates:
     * Inport: '<Root>/In1'
     */
    exprfld_B.S1 = exprfld_P.Gain_Gain * exprfld_U.i1;

    /* Gain: '<Root>/Gain1' incorporates:
     * Inport: '<Root>/In2'
     */
    exprfld_B.S2 = exprfld_P.Gain1_Gain * exprfld_U.i2;

    /* Output: '<Root>/Out1' incorporates:
     * Product: '<Root>/Product'
     */
    exprfld_Y.Out1 = exprfld_B.S1 * exprfld_B.S2;
}
```

There are separate code statements for both Gain blocks. Before final output, these code statements compute temporary results for the Gain blocks.

Enable Optimization

Expression folding is on by default. To see if expression folding is on for an existing model:

- 1 Open the Configuration Parameters dialog box and select the **All Parameters** tab.
- 2 Expression folding is available only when the **Signal storage reuse** parameter is set to **ON** because expression folding operates only on expressions involving local variables. On the **All Parameters** tab, select **Signal storage reuse**.
- 3 When you select **Signal storage reuse**, the **Enable local block outputs**, **Reuse local block outputs**, and **Eliminate superfluous local variables (expression folding)** parameters are all on by default.

Generate Code with Optimization

With expression folding, the code generator generates a single-line output computation, as shown in the `exprfld.c` file. The generated comments document the block parameters that appear in the expression.

```
/* Model step function */
void exprfld_step(void)
{
    /* Outport: '<Root>/Out1' incorporates:
     * Gain: '<Root>/Gain'
     * Gain: '<Root>/Gain1'
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     * Product: '<Root>/Product'
     */
    exprfld_Y.Out1 = exprfld_P.Gain_Gain * exprfld_U.i1 * (exprfld_P.Gain1_Gain *
        exprfld_U.i2);
}
```

For an example of expression folding in the context of a more complex model, click `rtwdemo_slexprfold`, or at the command prompt, type:

```
rtwdemo_slexprfold
```

For more information, see “Enable and Reuse Local Block Outputs in Generated Code” on page 31-102

See Also

“Signal storage reuse” | “Reuse local block outputs” | “Enable local block outputs” | “Eliminate superfluous local variables (Expression folding)”

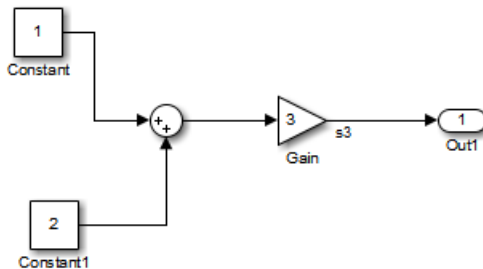
Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Control Signals and States in Code by Applying Storage Classes” on page 14-120

Inline Invariant Signals

You can optimize the generated code by selecting **Inline invariant signals** on the **Optimization > Signals and Parameters** pane. The generated code uses the numerical values of the invariant signals instead of their symbolic names.

An invariant signal is a block output signal that does not change during Simulink simulation. For example, the signal **S3** is an invariant signal. An *invariant signal* is not the same as an *invariant constant*. The two constants (1 and 2) and the gain value of 3 are invariant constants. To inline invariant constants, set **Default parameter behavior** to **Inlined**.



Optimize Generated Code Using Inline Invariant Signals

This example shows how to use inline invariant signals to optimize the generated code. This optimization transforms symbolic names of invariant signals into constant values.

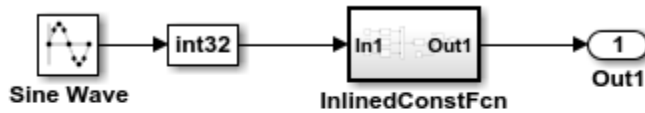
The `InlineInvariantSignals` optimization:

- Reduces ROM and RAM consumption.
- Improves execution speed.

Example Model

Consider the model `matlab:rtwdemo_inline_invariant_signals`.

```
model = 'rtwdemo_inline_invariant_signals';
open_system(model);
```



Copyright 2014 The MathWorks, Inc.

Generate Code

Create a temporary folder (in your system temporary folder) for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model using Simulink Coder.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_inline_invariant_signals
### Successful completion of build procedure for model: rtwdemo_inline_invariant_signals
```

View the generated code without the optimization. These lines of code are in `rtwdemo_inline_invariant_signals.c`.

```
cfile = fullfile(cgDir,'rtwdemo_inline_invariant_signals_grt_rtw',...
    'rtwdemo_inline_invariant_signals.c');
rtwdemodbtype(cfile,'/* Output and update for atomic system',...
    '/* Model output', 1, 0);

/* Output and update for atomic system: '<Root>/InlinedConstFcn' */
void rtwdemo_inline__InlinedConstFcn(int32_T rtu_In1,
    B_InlinedConstFcn_rtwdemo_inl_T *localB, const ConstB_InlinedConstFcn_rtwdem_T
    *localC)
{
    /* Product: '<S1>/Product' */
    localB->Product = rtu_In1 * localC->Sum_p;
}

```

Enable Optimization

- 1 Open the Configuration Parameters dialog box.
- 2 On the **Optimization->Signals and Parameters** pane, select **Inline Invariant Signals**.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model, 'InlineInvariantSignals', 'on');
```

Generate Code with Optimization

The generated code uses the numerical values of the folded constants instead of creating an additional structure (rtwdemo_inline_invariant_ConstB).

Build the model using Simulink Coder.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_inline_invariant_signals
### Successful completion of build procedure for model: rtwdemo_inline_invariant_signals
```

View the generated code with the optimization. These lines of code are in `rtwdemo_minmax.c`.

```
rtwdemodbtype(cfile, '/* Output and update for atomic system', '/* Model output', 1, 0)
```

```
/* Output and update for atomic system: '<Root>/InlinedConstFcn' */
void rtwdemo_inline__InlinedConstFcn(int32_T rtu_In1,
    B_InlinedConstFcn_rtwdemo_inl_T *localB)
{
    /* Product: '<S1>/Product' */
    localB->Product = rtu_In1 << 5;
}

```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

“Inline invariant signals”

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Inline Numeric Values of Block Parameters” on page 31-47

Inline Numeric Values of Block Parameters

This example shows how to optimize the generated code by inlining the numeric values of block parameters. *Block parameters* include the **Gain** parameter of a Gain block and the table data and breakpoint sets of an n-D Lookup Table block.

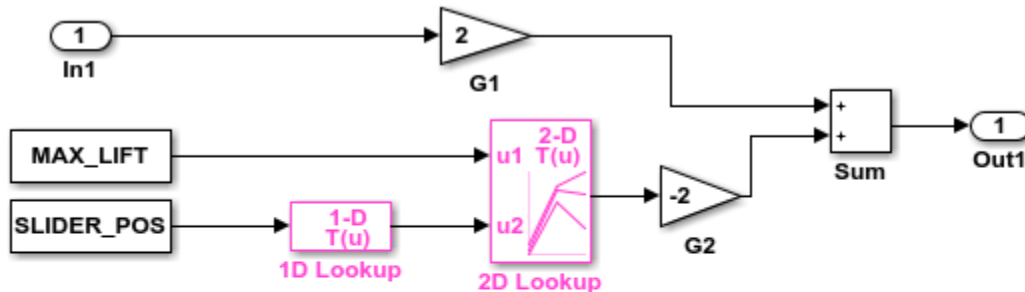
This optimization determines whether numeric block parameters occupy global memory in the generated code. The optimization can:

- Improve execution speed.
- Reduce RAM and ROM consumption.

Explore Example Model

Open the example model `rtwdemo_paraminline`.

```
open_system('rtwdemo_paraminline')
```



Generate Code Using Simulink Coder (double-click)	Generate Code Using Embedded Coder (double-click)	View Optimization Configuration (double-click)	Display Sample Time Colors (double-click)
--	--	---	--

Copyright 1994-2015 The MathWorks, Inc.

The model contains blocks that have these numeric parameters:

- The **Gain** parameters of the Gain blocks

- The **Constant value** parameters of the Constant blocks
- The table data and breakpoint sets of the n-D Lookup Table blocks

The output of the block G2, and the outputs of blocks upstream of G2, change only if you tune the values of the block parameters during simulation or during code execution. When you update the model diagram, these blocks and signal lines appear magenta in color.

Several blocks use `Simulink.Parameter` objects in the base workspace to set the values of their parameters. The parameter objects all use the storage class `Auto`, which means that you can configure the generated code to inline the parameter values.

Generate Code Without Optimization

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;  
[~,cgDir] = rtwdemodir();
```

Disable the optimization by setting **Configuration Parameters > Optimization > Signals and Parameters > Default parameter behavior** to **Tunable**.

```
set_param('rtwdemo_paraminline', 'DefaultParameterBehavior', 'Tunable')
```

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')  
  
### Starting build procedure for model: rtwdemo_paraminline  
### Successful completion of build procedure for model: rtwdemo_paraminline
```

In the code generation report, view the source file `rtwdemo_paraminline_data.c`. The code defines a global structure that contains the block parameter values. Each block parameter in the model, such as a lookup table array, breakpoint set, or gain, appears as a field of the structure.

```
cfile = fullfile(cgDir, 'rtwdemo_paraminline_grt_rtw', 'rtwdemo_paraminline_data.c');  
rtwdemodbtype(cfile, '/* Block parameters (auto storage) */', '};', 1, 1);
```

```
/* Block parameters (auto storage) */  
P_rtwdemo_paraminline_T rtwdemo_paraminline_P = {  
    10.0, /* Variable: MAX_LIFT  
          * Referenced by: '<Root>/Constant'  
          */
```



```

0.0,                                     /* Variable: SLIDER_POS
                                         * Referenced by: '<Root>/Constant1'
                                         */

/* Variable: T1Break
 * Referenced by: '<Root>/1D Lookup'
 */
{ -5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0 },

/* Variable: T1Data
 * Referenced by: '<Root>/1D Lookup'
 */
{ -1.0, -0.99, -0.98, -0.96, -0.76, 0.0, 0.76, 0.96, 0.98, 0.99, 1.0 },

/* Variable: T2Break
 * Referenced by: '<Root>/2D Lookup'
 */
{ 1.0, 2.0, 3.0 },

/* Variable: T2Data
 * Referenced by: '<Root>/2D Lookup'
 */
{ 4.0, 16.0, 10.0, 5.0, 19.0, 18.0, 6.0, 20.0, 23.0 },
2.0,                                     /* Expression: 2
                                         * Referenced by: '<Root>/G1'
                                         */
-2.0,                                     /* Expression: -2
                                         * Referenced by: '<Root>/G2'
                                         */

/* Computed Parameter: uDLookup_maxIndex
 * Referenced by: '<Root>/2D Lookup'
 */
{ 2U, 2U }
};

```

You can tune the structure fields during code execution because they occupy global memory. However, at each step of the generated algorithm, the code must calculate the output of each block, including the outputs of the block G2 and the upstream blocks. View the algorithm in the model step function in the file `rtwdemo_paraminline.c`.

```

cfile = fullfile(cgDir,'rtwdemo_paraminline_grt_rtw','rtwdemo_paraminline.c');
rtwdemodbtype(cfile,'/* Model step function */','/* Model initialize function */',1,0)

```

```

/* Model step function */
void rtwdemo_paraminline_step(void)
{
    /* Outport: '<Root>/Out1' incorporates:
     * Constant: '<Root>/Constant'
     * Constant: '<Root>/Constant1'
     * Gain: '<Root>/G1'
     * Gain: '<Root>/G2'
     * Inport: '<Root>/In1'
     * Lookup_n-D: '<Root>/1D Lookup'
     * Lookup_n-D: '<Root>/2D Lookup'
     * Sum: '<Root>/Sum'
     */
    rtwdemo_paraminline_Y.Out1 = rtwdemo_paraminline_P.G1_Gain *
        rtwdemo_paraminline_U.In1 + rtwdemo_paraminline_P.G2_Gain * look2_binlx
        (rtwdemo_paraminline_P.MAX_LIFT, look1_binlx
        (rtwdemo_paraminline_P.SLIDER_POS, rtwdemo_paraminline_P.T1Break,
        rtwdemo_paraminline_P.T1Data, 10U), rtwdemo_paraminline_P.T2Break,
        rtwdemo_paraminline_P.T2Break, rtwdemo_paraminline_P.T2Data,
        rtwdemo_paraminline_P.uDLookup_maxIndex, 3U);
}

```

Generate Code with Optimization

Set **Default parameter behavior** to **Inlined**.

```
set_param('rtwdemo_paraminline', 'DefaultParameterBehavior', 'Inlined')
```

Generate code from the model.

```
rtwbuild('rtwdemo_paraminline')
```

```
### Starting build procedure for model: rtwdemo_paraminline
```

```
### Successful completion of build procedure for model: rtwdemo_paraminline
```

In the code generation report, view the algorithm in the file `rtwdemo_paraminline.c`.

```
rtwdemodbtype(cfile, '/* Model step function */', '/* Model initialize function */', 1, 0)
```

```

/* Model step function */
void rtwdemo_paraminline_step(void)
{
    /* Outport: '<Root>/Out1' incorporates:
     * Gain: '<Root>/G1'

```

```

    * Inport: '<Root>/In1'
    * Sum: '<Root>/Sum'
    */
    rtwdemo_paraminline_Y.Out1 = 2.0 * rtwdemo_paraminline_U.In1 + 150.0;
}

```

The code does not allocate memory for block parameters or for parameter objects that use the storage class `Auto`. Instead, the code generator uses the parameter values from the model, and from the parameter objects, to calculate and inline the constant output of the block G2, `150.0`. The generator also inlines the value of the **Gain** parameter of the Gain block G1, `2.0`.

With the optimization, the generated code leaves out computationally expensive algorithmic code for blocks such as the lookup tables. The optimized code calculates the output of a block only if the output can change during execution. For this model, only the outputs of the Inport block In1, the Gain block G1, and the Sum block can change.

Close the model and the code generation report.

```

bdclose('rtwdemo_paraminline')
rtwdemoclean;
cd(currentDir)

```

Preserve Block Parameter Tunability

When you set **Default parameter behavior** to `Inlined`, you can preserve block parameter tunability by creating `Simulink.Parameter` objects for individual parameters. You can configure each object to appear in the code as a tunable field of the global parameter structure or as an individual global variable. You can change parameter values during code execution and interface the generated code with your own handwritten code. For more information, see “Block Parameter Representation in the Generated Code”.

Inline Invariant Signals

You can select the **Inline invariant signals** code generation option (which also places constant values in the generated code) only when you set **Default parameter behavior** to `Inlined`. See “Inline Invariant Signals”.

See Also

“Default parameter behavior”

Related Examples

- “Block Parameter Representation in the Generated Code” on page 14-43

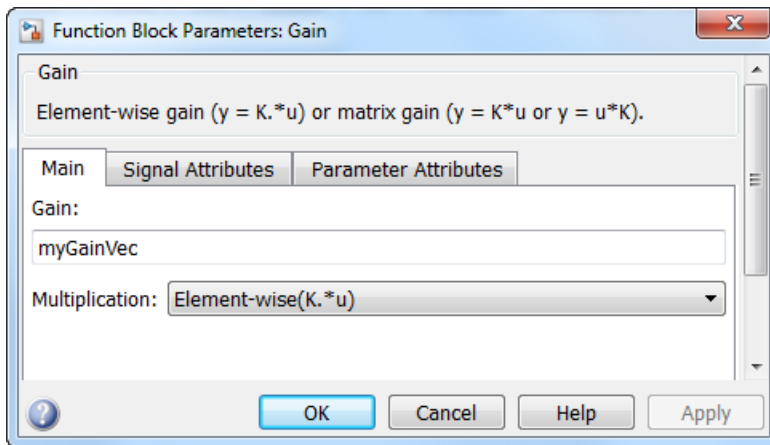
Configure Loop Unrolling Threshold

The **Loop unrolling threshold** parameter on the **Optimization > Signals and Parameters** pane determines when a wide signal or parameter should be wrapped into a `for` loop and when it should be generated as a separate statement for each element of the signal. The default threshold value is 5.

For example, consider the model below:



The gain parameter of the Gain block is the vector `myGainVec`.



Assume that the loop unrolling threshold value is set to the default, 5.

If `myGainVec` is declared as

```
myGainVec = [1:10];
```

an array of 10 elements, `myGainVec_P.Gain_Gain[]`, is declared within the `Parameters_model` data structure. The size of the gain array exceeds the loop unrolling threshold. Therefore, the code generated for the Gain block iterates over the array in a `for` loop, as shown in the following code:

```
{
    int32_T i1;

    /* Gain: '<Root>/Gain' */
    for(i1=0; i1<10; i1++) {
        myGainVec_B.Gain_f[i1] = rtb_foo *
            myGainVec_P.Gain_Gain[i1];
    }
}
```

If myGainVec is declared as

```
myGainVec = [1:3];
```

an array of three elements, myGainVec_P.Gain_Gain[], is declared within the Parameters data structure. The size of the gain array is below the loop unrolling threshold. The generated code consists of inline references to each element of the array, as in the code below.

```
/* Gain: '<Root>/Gain' */
myGainVec_B.Gain_f[0] = rtb_foo * myGainVec_P.Gain_Gain[0];
myGainVec_B.Gain_f[1] = rtb_foo * myGainVec_P.Gain_Gain[1];
myGainVec_B.Gain_f[2] = rtb_foo * myGainVec_P.Gain_Gain[2];
```

See “Explore Variable Names and Loop Rolling” for more information on loop rolling.

Note When a model includes Stateflow charts or MATLAB Function blocks, you can apply a set of Stateflow optimizations on the **Optimization > Stateflow** pane. The settings you select for the Stateflow options also apply to MATLAB Function blocks in the model. This is because the MATLAB Function blocks and Stateflow charts are built on top of the same technology and share a code base. You do not need a Stateflow license to use MATLAB Function blocks.

See Also

“Loop unrolling threshold”

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Optimize Generated Code by Combining Multiple for Constructs” on page 31-19

- “For Loop”

Use memcpy Function to Optimize Generated Code for Vector Assignments

In this section...
“Example Model” on page 31-57
“Generate Code” on page 31-58
“Generate Code with Optimization” on page 31-58

You can use the **Use memcpy for vector assignment** parameter to optimize generated code for vector assignments by replacing `for` loops with `memcpy` function calls. The `memcpy` function is more efficient than `for`-loop controlled element assignment for large data sets. This optimization improves execution speed.

Selecting the **Use memcpy for vector assignment** parameter enables the associated parameter **Memcpy threshold (bytes)**, which allows you to specify the minimum array size in bytes for which `memcpy` function calls should replace `for` loops in the generated code. For more information, see “Use memcpy for vector assignment” and “Memcpy threshold (bytes)”. In considering whether to use this optimization,

- Verify that your target supports the `memcpy` function.
- Determine whether your model uses signal vector assignments (such as `Y=expression`) to move large amounts of data, for example, using the Selector block.

To apply this optimization,

- 1 Consider first generating code without this optimization and measuring its execution speed, to establish a baseline for evaluating the optimized assignment.
- 2 Select **Use memcpy for vector assignment** and examine the setting of **Memcpy threshold (bytes)**, which by default specifies 64 bytes as the minimum array size for which `memcpy` function calls replace `for` loops. Based on the array sizes used in your application's signal vector assignments, and target environment considerations that might bear on the threshold selection, accept the default or specify another array size.
- 3 Generate code, and measure its execution speed against your baseline or previous iterations. Iterate on steps 2 and 3 until you achieve an optimal result.

Note: The `memcpy` optimization may not occur under certain conditions, including when other optimizations have a higher precedence than the `memcpy` optimization, or when the

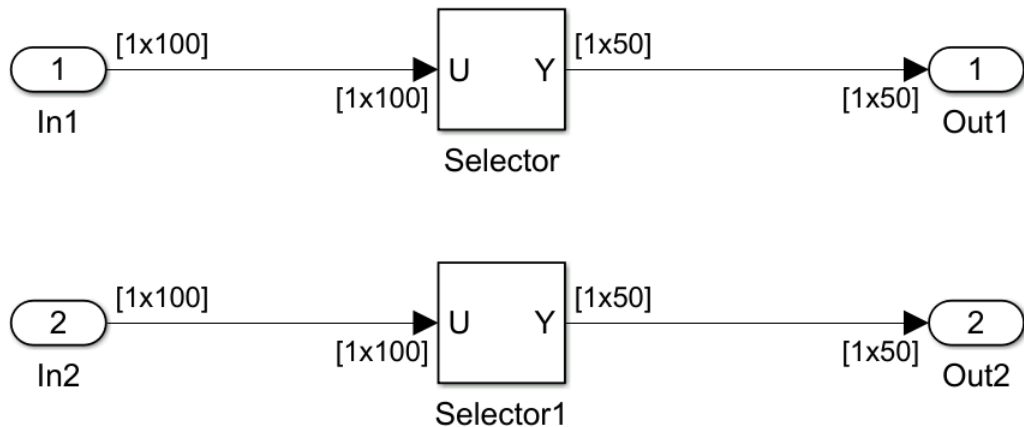
generated code is originating from Target Language Compiler (TLC) code, such as a TLC file associated with an S-function block.

Note: If you are licensed for Embedded Coder software, you can use a code replacement library (CRL) to provide your own custom implementation of the `memcpy` function to be used in generated model code. For more information, see “Memory Function Code Replacement”.

Example Model

To examine the result of using the **Use memcpy for vector assignment** parameter on the generated vector assignment code, create a model that generates signal vector assignments. For example,

- 1 Use In, Out, and Selector blocks to create the following model.



- 2 Open Model Explorer and configure the **Signal Attributes** for the In1 and In2 source blocks. For each, set **Port dimensions** to [1 , 100], and set **Data type** to `int32`. Apply the changes and save the model. In this example, the model has the name `vectorassign`.
- 3 For each Selector block, set the **Index** parameter to 1 : 50. Set the **Input port size** parameter to 100.

Generate Code

- 1 The **Use memcpy for vector assignment** parameter is on by default. To turn off the parameter, go to the **Optimization > Signals and Parameters** pane and clear the **Use memcpy for vector assignment** parameter.
- 2 Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the **Create code generation report**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model. When code generation completes, the HTML code generation report is displayed.
- 3 In the HTML code generation report, click the `vectorassign.c` section and inspect the model step function. Notice that the vector assignments are implemented using `for` loops.

```
/* Model step function */
void vectorassign_step(void)
{
    int32_T i;
    for (i = 0; i < 50; i++) {
        /* Output: '<Root>/Out1' incorporates:
         * Inport: '<Root>/In1'
         */
        vectorassign_Y.Out1[i] = vectorassign_U.In1[i];

        /* Output: '<Root>/Out2' incorporates:
         * Inport: '<Root>/In2'
         */
        vectorassign_Y.Out2[i] = vectorassign_U.In2[i];
    }
}
```

Generate Code with Optimization

- 1 Go to the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box and select the **Use memcpy for vector assignment** option. Leave the **Memcpy threshold (bytes)** option at its default setting of 64. Apply the changes and regenerate code for the model. When code generation completes, the HTML code generation report again is displayed.
- 2 In the HTML code generation report, click the `vectorassign.c` section and inspect the model output function. Notice that the vector assignments now are implemented using `memcpy` function calls.

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

```
/* Output: '<Root>/Out1' incorporates:  
 * Inport: '<Root>/In1'  
 */  
memcpy(&vectorassign_Y.Out1[0], &vectorassign_U.In1[0], 50U * sizeof(real_T));  
  
/* Output: '<Root>/Out2' incorporates:  
 * Inport: '<Root>/In2'  
 */  
memcpy(&vectorassign_Y.Out2[0], &vectorassign_U.In2[0], 50U * sizeof(real_T));  
}
```

See Also

“Use memcpy for vector assignment”

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Vector Operation Optimization” on page 31-99
- “Convert Data Copies to Pointer Assignments”

Generate Target Optimizations Within Algorithm Code

Some application components are hardware-specific and cannot simulate on a host system. For example, consider a component that includes pragmas and assembly code for enabling hardware instructions for saturate on add operations or a Fast Fourier Transform (FFT) function.

The following table lists integration options to customize generated algorithm code with target-specific optimizations.

Note: Solutions marked with *EC only* require an Embedded Coder license.

If...	Then...	For More Information, See
You want to optimize the execution speed and memory of the model code by replacing default math functions and operators with target-specific code	<i>EC only</i> —Implement function and operator replacements by using the Code Replacement Tool, code replacement library (CRL) API, and Code Replacement Viewer to create, examine, validate, and register hardware-specific replacement tables	“Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®”
You want to control how code generation technology declares, stores, and represents signals, tunable parameters, block states, and data objects in generated code	<i>EC only</i> —Design (create) and apply custom storage classes	<ul style="list-style-type: none"> • rtwdemo_cscpredef • rtwdemo_importstruct • rtwdemo_advsc • “Custom Storage Classes”

Note: To simulate an algorithm that includes target-specific elements in a host environment, you must create code that is equivalent to the target code and can run in the host environment.

Remove Code for Blocks That Have No Effect on Computational Results

This example shows how the code generator optimizes generated code by removing code that has no effect on computational results. This optimization:

- Increases execution speed.
- Reduces ROM consumption.

Example

In the model `rtwdemo_blockreduction`, a Gain block of value `1.0` is in between Inport and Outport blocks.

```
model = 'rtwdemo_blockreduction';
open_system(model);
```



Copyright 2014 The Mathworks, Inc.

Generate Code

Create a temporary folder for the build and inspection process.

```
currentDir=pwd;
[~,cgDir]=rtwdemodir();
```

Build the model.

```
set_param(model,'BlockReduction','off');
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_blockreduction
### Successful completion of build procedure for model: rtwdemo_blockreduction
```

Here is the code from `rtwdemo_blockreduction.c`.

```
cfile = fullfile(cgDir, 'rtwdemo_blockreduction_ert_rtw', 'rtwdemo_blockreduction.c');
rtwdemodbtype(cfile, '/* Model step function */', ...
    '/* Model initialize function */', 1, 0);

/* Model step function */
void rtwdemo_blockreduction_step(void)
{
    /* Outport: '<Root>/Out1' incorporates:
     * Gain: '<Root>/Gain'
     * Inport: '<Root>/In1'
     */
    rtwdemo_blockreduction_Y.Out1 = 1.0 * rtwdemo_blockreduction_U.In1;
}
```

Enable Optimization

- 1 Open the Configuration Parameters dialog box.
- 2 On the **All Parameters** tab, select **Block reduction**. This optimization is on by default.

Alternately, use the command-line API to enable the optimization.

```
set_param(model, 'BlockReduction', 'on');
```

Generate Code with Optimization

```
rtwbuild(model)

### Starting build procedure for model: rtwdemo_blockreduction
### Successful completion of build procedure for model: rtwdemo_blockreduction
```

Here is the optimized code from `rtwdemo_blockreduction.c`.

```
cfile = fullfile(cgDir, 'rtwdemo_blockreduction_ert_rtw', 'rtwdemo_blockreduction.c');
rtwdemodbtype(cfile, '/* Model step function */', ...
    '/* Model initialize function */', 1, 0);

/* Model step function */
void rtwdemo_blockreduction_step(void)
{
    /* Outport: '<Root>/Out1' incorporates:
```

```
* Inport: '<Root>/In1'  
*/  
rtwdemo_blockreduction_Y.Out1 = rtwdemo_blockreduction_U.In1;  
}
```

Because multiplying the input signal by a value of 1.0 does not impact computational results, the code generator excludes the Gain block from the generated code. Close the model and clean up.

```
bdclose(model)  
rtwdemoclean;  
cd(currentDir)
```

See Also

“Block reduction”

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Inline Numeric Values of Block Parameters” on page 31-47
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 31-40

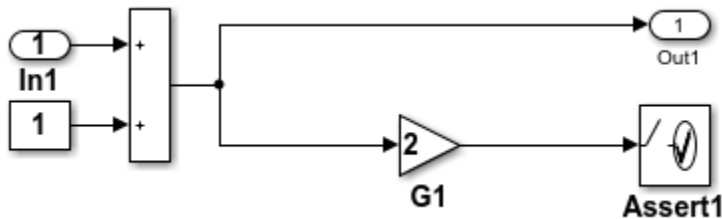
Eliminate Dead Code Paths in Generated Code

This example shows how the code generator eliminates dead (that is, unused) code paths from generated code. This optimization increases execution speed and conserves ROM and RAM consumption.

Example

In the model `rtwdemo_deadpathElim`, the signal leaving the Sum block divides into two separate code paths. The top path is not a dead code path. If the user disables the Assertion block, the bottom path becomes a dead code path.

```
model = 'rtwdemo_deadpathElim';
open_system(model);
```



Generate Code with an Enabled Assertion Block

- 1 For the Assertion block, open the block parameters dialog box.
- 2 Select the **Enable assertion** box. Alternatively, use the command-line API to enable the Assertion block.

```
set_param([model '/Assert1'], 'Enabled', 'on');
```

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_deadpathElim
```

```
### Successful completion of build procedure for model: rtwdemo_deadpathElim
```

Because the Assertion block is enabled, these lines of `rtwdemo_minmax.c` include code for the Gain and Assertion blocks.

```
cfile = fullfile(cgDir, 'rtwdemo_deadpathElim_grt_rtw', 'rtwdemo_deadpathElim.c');
rtwdemodbtype(cfile, /* Model step', /* Model initialize function */', 0, 1);
```

```
void rtwdemo_deadpathElim_step(void)
{
    /* Sum: '<Root>/Sum1' incorporates:
     * Constant: '<Root>/Constant1'
     * Inport: '<Root>/In1'
     */
    rtwdemo_deadpathElim_Y.Out1 = rtwdemo_deadpathElim_U.In1 + 1.0;

    /* Assertion: '<Root>/Assert1' incorporates:
     * Gain: '<Root>/G1'
     */
    utAssert(2.0 * rtwdemo_deadpathElim_Y.Out1 != 0.0);
}
```

Generate Code with a Disabled Assertion Block

Disable the Assertion block to generate a dead code path. The code generator detects the dead code path and eliminates it from the generated code.

- 1 For the Assertion block, open the Block Parameters dialog box.
- 2 Deselect the **Enable assertion** box.

Alternatively, use the command-line API to disable the Assertion block.

```
set_param([model '/Assert1'], 'Enabled', 'off');
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_deadpathElim
### Successful completion of build procedure for model: rtwdemo_deadpathElim
```

Because the Assertion block is disabled, these lines of `rtwdemo_minmax.c` do not include code for the Gain and Assertion blocks.

```
rtwdemodbtype(cfile, '/* Model step', '/* Model initialize function */', 0, 1);

void rtwdemo_deadpathElim_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   * Constant: '<Root>/Constant1'
   * Inport: '<Root>/In1'
   * Sum: '<Root>/Sum1'
   */
  rtwdemo_deadpathElim_Y.Out1 = rtwdemo_deadpathElim_U.In1 + 1.0;
}
```

Close the model and clean-up.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

For another example of how the code generator eliminates dead code paths in the generated code, see `rtwdemo_deadpath`.

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Remove Code for Blocks That Have No Effect on Computational Results” on page 31-62

Floating-Point Multiplication to Handle a Net Slope Correction

This example shows how to use floating-point multiplication to handle a net slope correction. When converting floating-point data types to fixed-point data types in the generated code, a net slope correction is one method of scaling fixed-point data types. Scaling the fixed-point data types avoids overflow conditions and minimizes quantization errors.

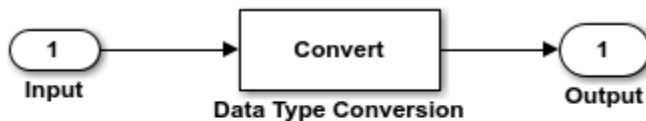
For processors that support efficient multiplication, using floating-point multiplication to handle a net slope correction improves code efficiency. If the net slope correction has a value that is not a power of two, using division improves precision.

Note: This example requires a Fixed-Point Designer™ license.

Example

In the model `rtwdemo_float_mul_for_net_slope_correction`, a Convert block converts an input signal from a floating-point data type to a fixed-point data type. The net slope correction has a value of 3.

```
model = 'rtwdemo_float_mul_for_net_slope_correction';  
open_system(model);
```



Copyright 2014 The MathWorks, Inc.

Generate Code

Create a temporary folder for the build and inspection process.

```
currentDir = pwd;  
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_float_mul_for_net_slope_correction
### Successful completion of build procedure for model: rtwdemo_float_mul_for_net_slope
```

In these lines of `rtwdemo_float_mul_for_net_slope_correction.c` code, the code generator divides the input signal by 3.0F .

```
cfile = fullfile(cgDir, 'rtwdemo_float_mul_for_net_slope_correction_ert_rtw', ...
    'rtwdemo_float_mul_for_net_slope_correction.c');
rtwdemodbtype(cfile, '/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_float_mul_for_net_slope_correction_step(void)
{
    /* Output: '<Root>/Output' incorporates:
     * DataTypeConversion: '<Root>/Data Type Conversion'
     * Inport: '<Root>/Input'
     */
    rtY.Output = (int16_T)(real32_T)floor((real_T)(rtU.Input / 3.0F));
}
```

Enable Optimization

- 1 Open the Configuration Parameters dialog box.
- 2 On the **Optimization** pane, select **Use floating-point multiplication to handle net slope corrections**. This optimization is on by default.

Alternatively, you can use the command-line API to enable the optimization.

```
set_param(model, 'UseFloatMulNetSlope', 'on');
```

Generate Code with Optimization

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_float_mul_for_net_slope_correction
### Successful completion of build procedure for model: rtwdemo_float_mul_for_net_slope
```

In the optimized code, the code generator multiplies the input signal by the reciprocal of 3.0F , that is 0.333333343F .

```
rtwdemodbtype(cfile, '/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_float_mul_for_net_slope_correction_step(void)
{
    /* Outport: '<Root>/Output' incorporates:
     * DataTypeConversion: '<Root>/Data Type Conversion'
     * Inport: '<Root>/Input'
     */
    rtY.Output = (int16_T)(real132_T)floor((real_T)(rtU.Input * 0.333333343F));
}
```

Close the model and the code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

“Use floating-point multiplication to handle net slope corrections”

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Remove Code From Floating-Point to Integer Conversions That Wraps Out-of-Range Values” on page 31-27
- “Subnormal Number Performance” on page 31-22

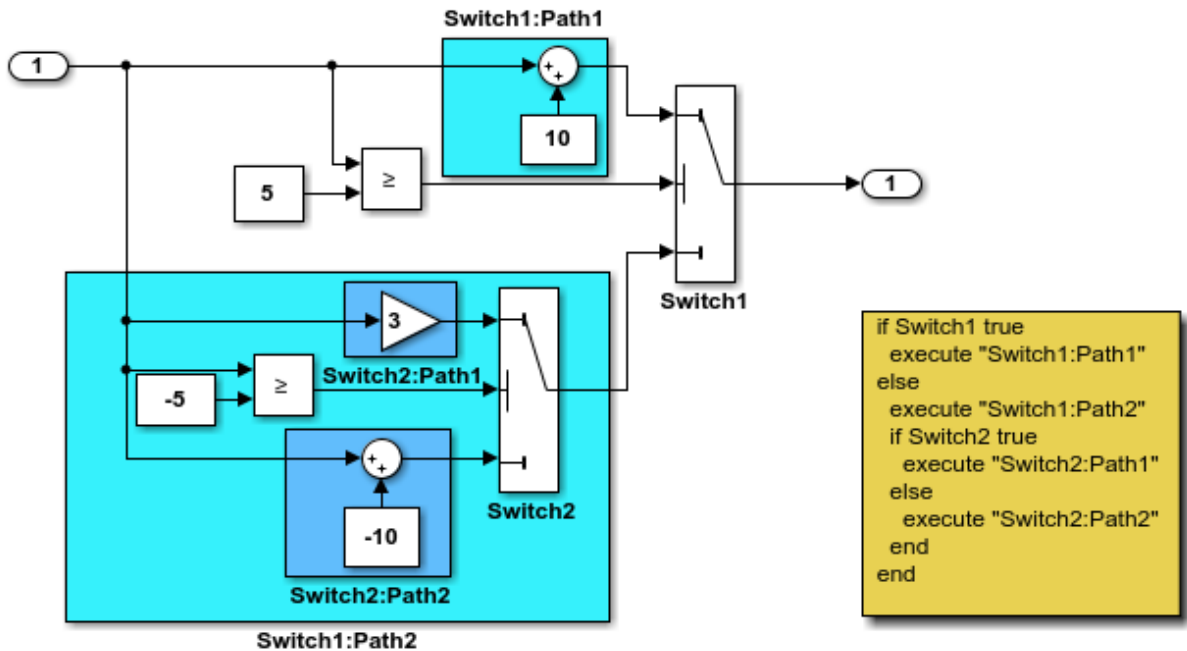
Use Conditional Input Branch Execution

This example shows how to optimize the generated code for a model that contains Switch and Multiport Switch blocks. When you select the model configuration parameter **Conditional input branch execution**, Simulink executes only blocks that compute the control input and data input that the control input selects. This optimization improves execution speed.

Example Model

In this example, switch paths are conditionally executed. If `Switch1` control input is true, `Switch1` executes blocks grouped in the `Switch1:Path1` branch. If `Switch1` control input is false, `Switch1` executes blocks grouped in the `Switch1:Path2` branch. If `Switch1` executes blocks in the `Switch1:Path2` branch and `Switch2` control input is true, `Switch2` executes blocks in the `Switch2:Path1` branch. If `Switch2` control input is false, `Switch2` executes blocks in the `Switch2:Path2` branch. The pseudo code shows this logic.

```
model='rtwdemo_condinput';  
open_system(model);
```



This model shows conditional input branch execution performed by Simulink and Simulink Coder. Conditional input branch execution improves simulation and code generation execution performance. In this example, switch paths are conditionally executed. That is, if Switch1's control input is true, Switch1 executes blocks grouped in the "Switch1:Path1" branch. Otherwise, it executes blocks grouped in the "Switch1:Path2" branch. If blocks in "Switch1:Path2" are executed, Switch2 executes the "Switch2:Path1" branch if its control input is true. Otherwise, it executes the "Switch2:Path2" branch. This logic is illustrated by the pseudo code above.

Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

Generate Code

The **Conditional input branch execution** parameter is on by default. Enter the following command-line API to turn off the parameter.

```
set_param(model, 'ConditionallyExecuteInputs', 'off');
```

Create a temporary folder for the build and inspection process.

```
currentDir=pwd;
[~,cgDir]=rtwdemodir();
```

Build the model.

```
rtwbuild(model)

### Starting build procedure for model: rtwdemo_condinput
### Successful completion of build procedure for model: rtwdemo_condinput
```

View the generated code without the optimization. These lines of code are in the `rtwdemo_condinput.c` file.

```
cfile = fullfile(cgDir,'rtwdemo_condinput_grt_rtw','rtwdemo_condinput.c');
rtwdemodbtype(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void rtwdemo_condinput_step(void)
{
  /* Switch: '<Root>/ Switch2' incorporates:
   * Constant: '<Root>/ C_10'
   * Constant: '<Root>/C_5'
   * Gain: '<Root>/ G3'
   * Inport: '<Root>/input'
   * RelationalOperator: '<Root>/Relational Operator'
   * Sum: '<Root>/ Sum'
   */
  if (rtwdemo_condinput_U.input >= -5.0) {
    rtwdemo_condinput_Y.output = 3.0 * rtwdemo_condinput_U.input;
  } else {
    rtwdemo_condinput_Y.output = rtwdemo_condinput_U.input + -10.0;
  }

  /* End of Switch: '<Root>/ Switch2' */
}
```

```

/* Switch: '<Root>/Switch1' incorporates:
 * Constant: '<Root>/C5'
 * Inport: '<Root>/input'
 * RelationalOperator: '<Root>/Relational Operator1'
 */
if (rtwdemo_condinput_U.input >= 5.0) {
  /* Outport: '<Root>/output' incorporates:
   * Constant: '<Root>/ C10'
   * Sum: '<Root>/ Sum1'
   */
  rtwdemo_condinput_Y.output = rtwdemo_condinput_U.input + 10.0;
}

/* End of Switch: '<Root>/Switch1' */
}

```

The generated code contains an **if-else** statement for the **Switch1** block and an **if** statement for the **Switch2** block. Therefore, the generated code for **Switch1:Path2** executes even if the **if** statement for **Switch1:Path1** evaluates to true.

Enable Optimization

- 1 Open the Configuration Parameters dialog box.
- 2 On the **All Parameters** tab, select **Conditional input branch execution**.
Alternatively, you can use the command-line API to enable the optimization.

```
set_param(model, 'ConditionallyExecuteInputs', 'on');
```

Generate Code with Optimization

```

rtwbuild(model)
cfile = fullfile(cgDir, 'rtwdemo_condinput_grt_rtw', 'rtwdemo_condinput.c');
rtwdemodbtype(cfile, '/* Model step', '/* Model initialize', 1, 0);

### Starting build procedure for model: rtwdemo_condinput
### Successful completion of build procedure for model: rtwdemo_condinput

/* Model step function */
void rtwdemo_condinput_step(void)
{
  /* Switch: '<Root>/Switch1' incorporates:
   * Constant: '<Root>/C5'
   * Constant: '<Root>/C_5'
   * Inport: '<Root>/input'

```

```

* RelationalOperator: '<Root>/Relational Operator'
* RelationalOperator: '<Root>/Relational Operator1'
* Switch: '<Root>/ Switch2'
*/
if (rtwdemo_condinput_U.input >= 5.0) {
/* Outport: '<Root>/output' incorporates:
* Constant: '<Root>/ C10'
* Sum: '<Root>/ Sum1'
*/
rtwdemo_condinput_Y.output = rtwdemo_condinput_U.input + 10.0;
} else if (rtwdemo_condinput_U.input >= -5.0) {
/* Outport: '<Root>/output' incorporates:
* Gain: '<Root>/ G3'
* Switch: '<Root>/ Switch2'
*/
rtwdemo_condinput_Y.output = 3.0 * rtwdemo_condinput_U.input;
} else {
/* Outport: '<Root>/output' incorporates:
* Constant: '<Root>/ C_10'
* Sum: '<Root>/ Sum'
* Switch: '<Root>/ Switch2'
*/
rtwdemo_condinput_Y.output = rtwdemo_condinput_U.input + -10.0;
}

/* End of Switch: '<Root>/Switch1' */
}

```

The generated code contains one `if` statement. The generated code for `Switch1:Path2` only executes if the `if` statement evaluates to false.

Close Model and Code Generation Report

```

bdclose(model)
rtwdemoclean;
cd(currentDir)

```

See Also

“Conditional input branch execution” | Multiport Switch | Switch

Related Examples

- “Optimization Tools and Techniques” on page 31-13

- “Eliminate Dead Code Paths in Generated Code” on page 31-65

Optimize Generated Code for Complex Signals

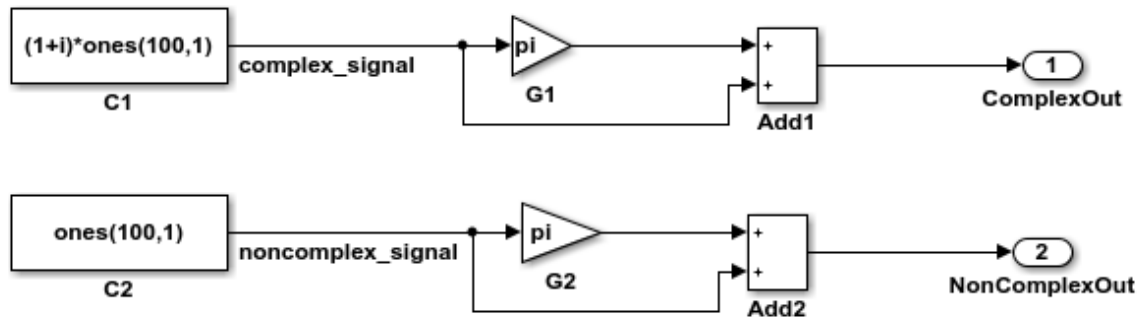
This example shows how Simulink Coder handles complex signals efficiently. To view the data types of the signals, update the model using **Simulation > Update Diagram**. Complex signals are represented as structures in generated code. Simulink Coder performs various optimizations on these structures. For example:

- Expression Folding: Gain and Sum operations on the complex signal are folded into a single expression.
- For-loop fusion: Two separate for-loops, one for the complex signal and one for noncomplex signal, are combined into a single for-loop.
- Inlined block parameters: The value of Gain block "pi" is inlined in the expression of the complex Gain-Sum.

Because of optimizations such as these, the code generated for complex and noncomplex signals is equally efficient.

Example Model

```
model='rtwdemo_complex';  
open_system(model);
```



This model shows how Simulink Coder handles complex signals efficiently. To view the data types of the signals, update the model using Simulation > Update Diagram. Complex signals are represented as structures in generated code. Simulink Coder performs various optimizations on these structures. For example,

- o Expression folding: Gain and Sum operations on the complex signal are folded into a single expression.
- o For-loop fusion: Two separate for-loops, one for the complex signal and one for the noncomplex signal, are combined into a single for-loop.
- o Inlined block parameters: The value of Gain block "pi" is inlined in the expression of the complex Gain-Sum.

Because of optimizations such as these, the code generated for complex and noncomplex signals is equally efficient.

**Generate Code Using
Simulink Coder
(double-click)**

**Generate Code Using
Embedded Coder
(double-click)**

Copyright 1994-2012 The MathWorks, Inc.

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 31-40
- “Disable Nonfinite Checks or Inlining for Math Functions” on page 31-34

Speed Up Linear Algebra in Code Generated from a MATLAB Function Block

To improve the execution speed of code generated for certain linear algebra functions in a MATLAB Function block, specify that the code generator produce LAPACK calls. LAPACK is a software library for numerical linear algebra. The code generator uses the LAPACKE C interface to LAPACK. If you specify that you want to generate LAPACK calls, and the input arrays for the linear algebra functions meet certain criteria, the code generator produces the LAPACK calls. Otherwise, the code generator produces code for the linear algebra functions.

The code generator uses the LAPACK library that you specify. Specify a LAPACK library that is optimized for your execution environment. See www.netlib.org/lapack/faq.html#_what_and_where_are_the_lapack_vendors_implementations.

Specify LAPACK Library

To generate LAPACK calls, you must have access to a LAPACK callback class. A LAPACK callback class specifies the LAPACK library and LAPACKE header file for the LAPACK calls. To indicate that you want to generate LAPACK calls and that you want to use a specific LAPACK library, specify the name of the LAPACK callback class. In the Configuration Parameters dialog box, in the **Code Generation** category, on the **All Parameters** tab, set **Custom LAPACK library callback** to the name of the callback class, for example, `useMyLAPACK`.

Write LAPACK Callback Class

To specify the locations of a particular LAPACK library and LAPACKE header file, write a LAPACK callback class. Share the callback class with others who want to use this LAPACK library for LAPACK calls in generated code.

The callback class must derive from the abstract class `coder.LAPACKCallback`. Use the following example callback class as a template.

```
classdef useMyLAPACK < coder.LAPACKCallback
    methods (Static)
        function hn = getHeaderFilename()
            hn = 'mylapacke_custom.h';
        end
    end
end
```

```

function updateBuildInfo(buildInfo, buildctx)
    buildInfo.addIncludePaths(fullfile(pwd, 'include'));
    libName = 'mylapack';
    libPath = fullfile(pwd, 'lib');
    [~, linkLibExt] = buildctx.getStdLibInfo();
    buildInfo.addLinkObjects([libName linkLibExt], libPath, ...
        '', true, true);
    buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
    buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
end
end
end

```

You must provide the `getHeaderFilename` and `updateBuildInfo` methods. The `getHeaderFilename` method returns the LAPACK header file name. In the example callback class, replace `mylapack_custom.h` with the name of your LAPACK header file. The `updateBuildInfo` method provides the information required for the build process to link to the LAPACK library. Use code like the code in the template to specify the location of header files and the full path name of the LAPACK library. In the example callback class, replace `mylapack` with the name of your LAPACK library.

If your compiler supports only complex data types that are represented as structures, include these lines in the `updateBuildInfo` method.

```

buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');

```

Generate LAPACK Calls by Specifying a LAPACK Callback Class

This example shows how to generate code that calls LAPACK functions in a specific LAPACK library. For this example, assume that the LAPACK callback class `useMyLAPACK` specifies the LAPACK library that you want.

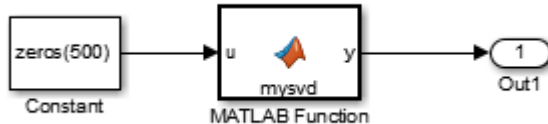
- 1 Create a Simulink model.
- 2 Add a MATLAB Function block to the model.
- 3 In the MATLAB Function block, add code that calls a linear algebra function. For example, add the function `mysvd` that calls the MATLAB function `svd`.

```

function s = mysvd(A)
    %#codegen
    s = svd(A);
end

```


- 4 Add a Constant block to the left of the MATLAB Function block. Set the value to `zeros(500)`.
- 5 Add an Outport block to the right of the MATLAB Function block.
- 6 Connect the blocks.



- 7 In the Configuration Parameters dialog box, in the **Code Generation** category, on the **All Parameters** tab, set **Custom LAPACK library callback** to `useMyLAPACK`.

The callback class must be on the MATLAB path.

- 8 Build the model.

If the input to `mysvd` is large enough, the code generator produces a LAPACK call for `svd`. Here is an example of a call to the LAPACK library function for `svd`.

```
info_t = LAPACKE_dgesvd(LAPACK_COL_MAJOR, 'N', 'N', (lapack_int)500,
    (lapack_int)500, &A[0], (lapack_int)500, &S[0], NULL, (lapack_int)1, NULL,
    (lapack_int)1, &superb[0]);
```

Locate LAPACK Library in Execution Environment

The LAPACK library must be available in your execution environment. If your LAPACK library is shared, use environment variables or linker options to specify the location of the LAPACK library.

- On a Windows platform, modify the `PATH` environment variable.
- On a Linux platform, modify the `LD_LIBRARY_PATH` environment variable or use the `rpath` linker option.
- On a Mac OS X, modify the `DYLD_LIBRARY_PATH` environment variable or use the `rpath` linker option.

To specify the `rpath` linker option, you can use the build information `addLinkFlags` method in the `updateBuildInfo` method of your `Coder.LAPACKCallback` class. For example, for a GCC compiler:

```
buildInfo.addLinkFlags(sprintf('-Wl,-rpath,"%s"', libPath));
```

See Also

`coder.LAPACKCallback`

More About

- “LAPACK Calls for Linear Algebra in a MATLAB Function Block”

External Websites

- www.netlib.org/lapack
- www.netlib.org/lapack/faq.html#_what_and_where_are_the_lapack_vendors_implementations

Optimize Memory Usage for Time Counters

This example shows how to optimize the amount of memory that the code generator allocates for time counters. The example optimizes the memory that stores elapsed time, the interval of time between two events.

The code generator represents time counters as unsigned integers. The word size of time counters is based on the setting of the model configuration parameter **Application lifespan (days)**, which specifies the expected maximum duration of time the application runs. You can use this parameter to prevent time counter overflows. The default size is 64 bits.

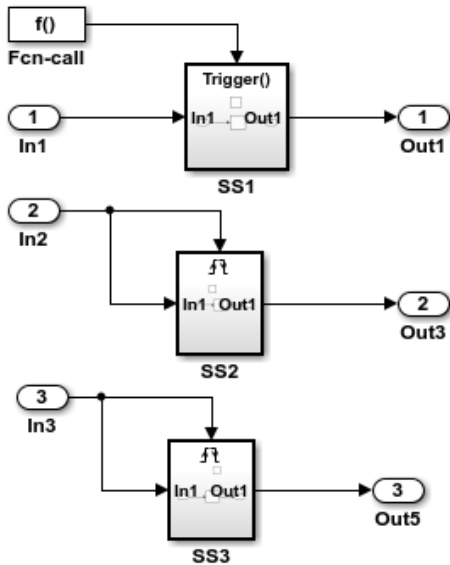
The number of bits that a time counter uses depends on the setting of the **Application lifespan (days)** parameter. For example, if a time counter increments at a rate of 1 kHz, to avoid an overflow, the counter has the following number of bits:

- Lifespan < 0.25 sec: 8 bits
- Lifespan < 1 min: 16 bits
- Lifespan < 49 days: 32 bits
- Lifespan > 50 days: 64 bits

A 64-bit time counter does not overflow for 590 million years.

Open Example Model

Open the example model `rtwdemo_abstime`.



SS1 is clocked at 1 kHz, and contains a discrete-time integrator that requires elapsed time to compute its output. However, a counter is not required to compute elapsed time since the trigger port 'Sample time type' is set to 'periodic.' Instead, time is inlined as 1 kHz.

SS2 is clocked at 100 Hz, and contains a discrete-time integrator that requires elapsed time to compute its output. Since the application life span of the model is 1 day, a 32-bit counter is required to compute elapsed time for SS2.

SS3 is clocked at 0.5 Hz, and contains a discrete-time integrator that requires elapsed time to compute its output. Since the application life span of the model is 1 day, a 16-bit counter is required to compute elapsed time for SS3.

Simulink Coder optimizes how counters are employed to measure absolute and elapsed time:

- o Time is computed from unsigned integer counters.
- o Only tasks that require time are allocated a counter.
- o Elapsed time is computed by a subsystem if and only if a block in its hierarchy requires elapsed time.
- o Time is shared by all blocks within a triggered hierarchy.

Simulink Coder further optimizes counters based on the option "Application life span," whereby the number of bits used for a particular counter is optimized based on how long the application will run.

Did you know ...

Display Sample Time Colors (double-click)

Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

Copyright 1994-2012 The MathWorks, Inc.

The model consists of three subsystems SS1, SS2, and SS3. On the **Optimization** tab, the **Application lifespan (days)** parameter is set to the default, which is **auto**.

The three subsystems contain a discrete-time integrator that requires elapsed time as input to compute its output value. The subsystems vary as follows:

- SS1 - Clocked at 1 kHz. Does not require a time counter. **Sample time type** parameter for trigger port is set to `periodic`. Elapsed time is inlined as 0.001.
- SS2 - Clocked at 100 Hz. Requires a time counter. Based on a lifespan of 1 day, a 32-bit counter stores the elapsed time.
- SS3 - Clocked at 0.5 Hz. Requires a time counter. Based on a lifespan of 1 day, a 16-bit counter stores the elapsed time.

Simulate the Model

Simulate the model. By default, the model is configured to show sample times in different colors. Discrete sample times for the three subsystems appear red, green, and blue. Triggered subsystems are blue-green.

Generate Code and Report

1. Create a temporary folder for the build and inspection process.
2. Configure the model for the code generator to use the GRT system target file and a lifespan of `inf` days.
3. Build the model.

```
### Starting build procedure for model: rtwdemo_abstime
### Successful completion of build procedure for model: rtwdemo_abstime
```

Review Generated Code

Open the generated source file `rtwdemo_abstime.c`.

```
struct tag_RTM_rtwdemo_abstime_T {
    const char_T *errorStatus;

    /*
     * Timing:
     * The following substructure contains information regarding
     * the timing information for the model.
     */
    struct {
        uint32_T clockTick1;
        uint32_T clockTickH1;
        uint32_T clockTick2;
        uint32_T clockTickH2;
        struct {
```

```
        uint16_T TID[3];
        uint16_T cLimit[3];
    } TaskCounters;
} Timing;
};
```

Four 32-bit unsigned integers, `clockTick1`, `clockTickH1`, `clockTick2`, and `clockTickH2` are counters for storing the elapsed time of subsystems `SS2` and `SS3`.

Enable Optimization and Regenerate Code

1. Reconfigure the model to set the lifespan to 1 day.
2. Build the model.

```
### Starting build procedure for model: rtwdemo_abstime
### Successful completion of build procedure for model: rtwdemo_abstime
```

Review the Regenerated Code

```
struct tag_RTM_rtwdemo_abstime_T {
    const char_T *errorStatus;

    /*
     * Timing:
     * The following substructure contains information regarding
     * the timing information for the model.
     */
    struct {
        uint32_T clockTick1;
        uint16_T clockTick2;
        struct {
            uint16_T TID[3];
            uint16_T cLimit[3];
        } TaskCounters;
    } Timing;
};
```

The new setting for the **Application lifespan (days)** parameter instructs the code generator to set aside less memory for the time counters. The regenerated code includes:

- 32-bit unsigned integer, `clockTick1`, for storing the elapsed time of the task for `SS2`

- 16-bit unsigned integer, `clockTick2`, for storing the elapsed time of the task for SS3

Related Information

- “Optimization Pane: General”
- “Timers in Asynchronous Tasks”
- “Time-Based Scheduling and Code Generation”

More About

- “Optimization Tools and Techniques” on page 31-13
- “Control Memory Allocation for Time Counters” on page 31-15
- “Access Timers Programmatically” on page 3-5
- “Generate Code for an Elapsed Time Counter” on page 3-9
- “Absolute Time Limitations” on page 3-12

Minimize Memory Requirements During Code Generation

When the code generator produces code, it creates an partial representation of your model (called *model.rtw*), which the Target Language Compiler parses to transform block computations, parameters, signals, and constant data into a high-level language, (for example, C). Parameters and data are normally copied into the *model.rtw* file, whether they originate in the model itself or come from variables or objects in a workspace.

Models which have large amounts of parameter and constant data (such as lookup tables) can tax memory resources and slow down code generation because of the need to copy their data to *model.rtw*. You can improve code generation speed by limiting the size of data that is copied by using a `set_param` command, described below.

Data vectors such as those for parameters, lookup tables, and constant blocks whose sizes exceed a specified value are not copied into the *model.rtw* file. In place of the data vectors, the code generator places a special reference key in the *model.rtw* file that enables the Target Language Compiler to access the data directly from the Simulink software and format it directly into the generated code. This results in maintaining only one copy of large data vectors in memory.

You can specify the maximum number of elements that a parameter or other data source can have for the code generator to represent it literally in the *model.rtw* file. Whenever this threshold size is exceeded, the product writes a reference to the data to the *model.rtw* file, rather than its values. The default threshold value is 10 elements, which you can verify with

```
get_param(0, 'RTWDataReferencesMinSize')
```

To set the threshold to a different value, type the following `set_param` function in the MATLAB Command Window:

```
set_param(0, 'RTWDataReferencesMinSize', <size>)
```

Provide an integer value for `size` that specifies the number of data elements above which reference keys are to be used in place of actual data values.

Related Examples

- “Increase Code Generation Speed” on page 31-7

Optimize Generated Code Using Boolean Data for Logical Signals

Optimize generated code by storing logical signals as Boolean data. When you select the model configuration parameter **Implement logic signals as Boolean data (vs. double)**, blocks that generate logic signals output Boolean signals.

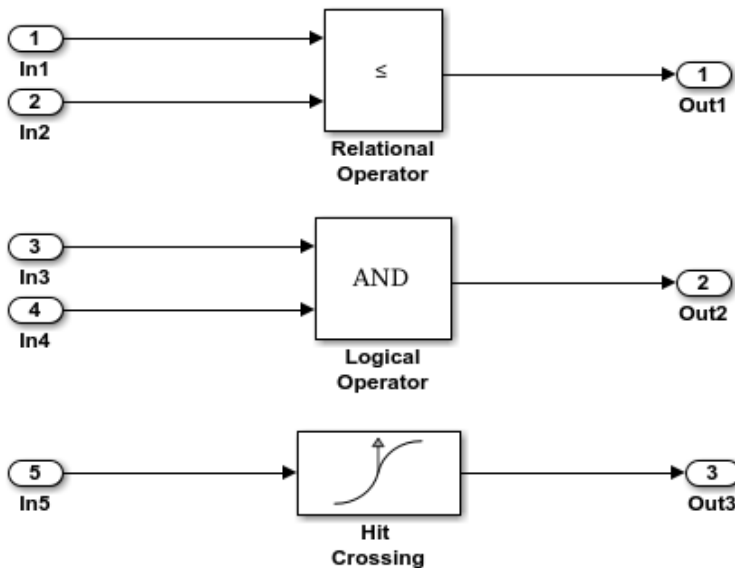
The optimization:

- Reduces the ROM and RAM consumption.
- Improves execution speed.

Example Model

Consider the model `rtwdemo_logicalAsBoolean`. The outputs of the Relational Operator, Logical Operator and Hit Crossing blocks are double, even though they represent logical data.

```
model = 'rtwdemo_logicalAsBoolean';  
open_system(model);
```



Copyright 1994-2012 The MathWorks, Inc.

Generate Code

Create a temporary folder (in your system temporary folder) for the build and inspection process.

```
currentDir = pwd;  
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_logicalAsBoolean  
### Successful completion of build procedure for model: rtwdemo_logicalAsBoolean
```

View the generated code without the optimization. These lines of code are in `rtwdemo_logicalAsBoolean.h`.

```
hfile = fullfile(cgDir,'rtwdemo_logicalAsBoolean_ert_rtw',...  
    'rtwdemo_logicalAsBoolean.h');  
rtwdemodbtype(hfile,'/* External outputs','/* Parameters (auto storage) */',1,0);  
  
/* External outputs (root outputs fed by signals with auto storage) */  
typedef struct {  
    real_T Out1;           /* '<Root>/Out1' */  
    real_T Out2;           /* '<Root>/Out2' */  
    real_T Out3;           /* '<Root>/Out3' */  
} ExtY_rtwdemo_logicalAsBoolean_T;
```

Enable Optimization

- 1 Open the Configuration Parameters dialog box.
- 2 On the **All Parameters** tab, select **Implement logic signals as Boolean data (vs. double)**.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model,'BooleanDataType','on');
```

Generate Code with Optimization

The generated code stores the logical signal output as Boolean data.

Build the model.

```
rtwbuild(model)

### Starting build procedure for model: rtwdemo_logicalAsBoolean
### Successful completion of build procedure for model: rtwdemo_logicalAsBoolean

View the generated code with the optimization. These lines of code are in
rtwdemo_logicalAsBoolean.h.

rtwdemodbtype(hfile, '/* External outputs', '/* Parameters (auto storage) */', 1, 0);

/* External outputs (root outputs fed by signals with auto storage) */
typedef struct {
    boolean_T Out1;           /* '<Root>/Out1' */
    boolean_T Out2;           /* '<Root>/Out2' */
    boolean_T Out3;           /* '<Root>/Out3' */
} ExtY_rtwdemo_logicalAsBoolean_T;
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

“Implement logic signals as Boolean data (vs. double)”

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Data Types Supported by Simulink”
- “Use Conditional Input Branch Execution” on page 31-71
- “Bitfields”

Reduce Memory Usage for Boolean and State Configuration Variables

- 1 Open the Model Configuration Parameters dialog box.
- 2 In the Model Configuration Parameters dialog box, select the **Optimization > Stateflow** pane.
- 3 Choose from these options:
 - **Use bitsets for storing state configuration** — Reduces the amount of memory that stores state configuration variables. However, it can increase the amount of memory that stores target code if the target processor does not include instructions for manipulating bitsets.
 - **Use bitsets for storing Boolean data** — Reduces the amount of memory that stores Boolean variables. However, it can increase the amount of memory that stores target code if the target processor does not include instructions for manipulating bitsets.

Note: You cannot use bitsets when you generate code for these cases:

- An external mode simulation
 - A target that specifies an explicit structure alignment
-

See Also

“Use bitsets for storing state configuration” | “Use bitsets for storing Boolean data”

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Design Tips for Optimizing Generated Code for Stateflow Objects” on page 31-10
- “Optimize Generated Code Using Boolean Data for Logical Signals” on page 31-89
- “Bitfields”

Customize Stack Space Allocation

Your application might be constrained by limited memory. Controlling the maximum allowable size for the stack is one way to modify whether data is defined as local or global in the generated code. You can limit the use of stack space by specifying a positive integer value for the “Maximum stack size (bytes)” parameter, on the **Optimization > Signals and Parameters** pane of the Configuration parameter dialog box. Specifying the maximum allowable stack size provides control over the number of local and global variables in the generated code. Specifically, lowering the maximum stack size might generate more variables into global structures. The number of local and global variables help determine the required amount of stack space for execution of the generated code.

The default setting for “Maximum stack size (bytes)” is **Inherit from target**. In this case, the value of the maximum stack size is the smaller value of the following: the default value set by the code generator (200,000 bytes) or the value of the TLC variable `MaxStackSize` found in the system target file (`ert.tlc`).

To specify a smaller stack size for your application, select the **Specify a value** option of the **Maximum stack size (bytes)** parameter and enter a positive integer value. To specify a smaller stack size at the command line, use:

```
set_param(model_name, 'MaxStackSize', 65000);
```

Note: For overall executable stack usage metrics, you might want to do a target-specific measurement, such as using runtime (empirical) analysis or static (code path) analysis with object code.

It is recommended that you use the **Maximum stack size (bytes)** parameter to control stack space allocation instead of modifying the TLC variable, `MaxStackSize`, in the system target file. However, a target author might want to set the TLC variable, `MaxStackSize`, for a target. To set `MaxStackSize`, use `assign` statements in the system target file (`ert.tlc`), as in the following example.

```
%assign MaxStackSize = 4096
```

Write your `%assign` statements in the **Configure RTW code generation settings** section of the system target file. The `%assign` statement is described in “Target Language Compiler”.

See Also

“Maximum stack size (bytes)”

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Enable and Reuse Local Block Outputs in Generated Code” on page 31-102
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 31-40
- “Inline Numeric Values of Block Parameters” on page 31-47

Optimize Generated Code Using memset Function

This example shows how to optimize the generated code by using the `memset` function to clear the internal storage. When you select the model configuration parameter **Use memset to initialize floats and doubles to 0.0**, the `memset` function clears internal storage, regardless of type, to the integer bit pattern 0 (that is, all bits are off).

If your compiler and target CPU both represent floating-point zero with the integer bit pattern 0, consider setting this parameter to gain execution and ROM efficiency.

NOTE: The command-line values are the reverse of the settings values. 'on' in the command line corresponds to clearing the setting. 'off' in the command line corresponds to selecting the setting.

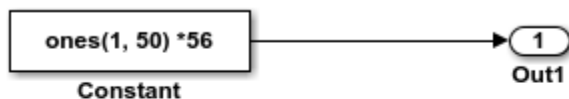
This optimization:

- Reduces ROM consumption.
- Improves execution speed.

Example Model

Consider the model `matlab:rtwdemo_memset`.

```
model = 'rtwdemo_memset';  
open_system(model);
```



Copyright 2014 The MathWorks, Inc.

Generate Code

The code generator uses a loop to initialize the Constant block values.

Create a temporary folder (in your system temporary folder) for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwdemodir();
```

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_memset
### Successful completion of build procedure for model: rtwdemo_memset
```

View the generated code without the optimization. These lines of code are in `rtwdemo_memset.c`.

```
cfile = fullfile(cgDir, 'rtwdemo_memset_grt_rtw', 'rtwdemo_memset.c');
rtwdemodbtype(cfile, '/* Model initialize function */', ...
    '/* Model terminate function */', 1, 0);
```

```
/* Model initialize function */
void rtwdemo_memset_initialize(void)
{
    /* Registration code */

    /* initialize error status */
    rtmSetErrorStatus(rtwdemo_memset_M, (NULL));

    /* external outputs */
    {
        int32_T i;
        for (i = 0; i < 50; i++) {
            rtwdemo_memset_Y.Out1[i] = 0.0;
        }
    }

    {
        int32_T i;

        /* ConstCode for Outport: '<Root>/Out1' */
        for (i = 0; i < 50; i++) {
            rtwdemo_memset_Y.Out1[i] = 56.0;
        }

        /* End of ConstCode for Outport: '<Root>/Out1' */
    }
}
```


Enable Optimization

- 1 Open the Configuration Parameters dialog box.
- 2 On the **All Parameters** tab, select **Use memset to initialize floats and doubles to 0.0**.

Alternatively, you can use the command-line API to enable the optimization:

```
set_param(model, 'InitFltsAndDblsToZero', 'off');
```

Generate Code with Optimization

The code generator uses the `memset` function to initialize the Constant block values.

Build the model.

```
rtwbuild(model)
```

```
### Starting build procedure for model: rtwdemo_memset
### Successful completion of build procedure for model: rtwdemo_memset
```

View the generated code with the optimization. These lines of code are in `rtwdemo_memset.c`.

```
rtwdemodbtype(cfile, '/* Model initialize function */', ...
              '/* Model terminate function */', 1, 0);
```

```
/* Model initialize function */
void rtwdemo_memset_initialize(void)
{
    /* Registration code */

    /* initialize error status */
    rtmSetErrorStatus(rtwdemo_memset_M, (NULL));

    /* external outputs */
    (void) memset(&rtwdemo_memset_Y.Out1[0], 0,
                 50U*sizeof(real_T));

    {
        int32_T i;

        /* ConstCode for Output: '<Root>/Out1' */
        for (i = 0; i < 50; i++) {
```

```
        rtwdemo_memset_Y.Out1[i] = 56.0;
    }

    /* End of ConstCode for Outport: '<Root>/Out1' */
}
}
```

Close the model and the code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

See Also

“Use memset to initialize floats and doubles to 0.0”

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Use memcpy Function to Optimize Generated Code for Vector Assignments” on page 31-56
- “Vector Operation Optimization” on page 31-99
- “Remove Initialization Code”

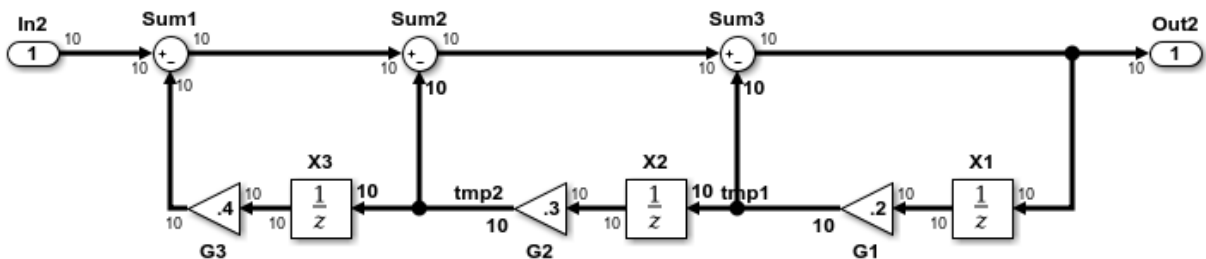
Vector Operation Optimization

This example shows how Simulink® Coder™ optimizes generated code by setting block output that generates vectors to scalars, for blocks such as the Mux, Sum, Gain, and Bus. This optimization reduces stack memory by replacing temporary local arrays with local variables.

Example Model

In the model, `rtwdemo_VectorOptimization`, the output of Gain blocks G1 and G2 are the vector signals `tmp1` and `tmp2`. These vectors have a width of 10.

```
model = 'rtwdemo_VectorOptimization';
open_system(model);
set_param(model, 'SimulationCommand', 'update')
```



Copyright 2014 The MathWorks, Inc.

Generate Code

Create a temporary folder (in your system temporary folder) for the build and inspection process.

```
currentDir = pwd;
[~,cgDir] = rtwmoddir();
```

Build the model.

```
rtwbuild(model)
```

```

### Starting build procedure for model: rtwdemo_VectorOptimization
### Successful completion of build procedure for model: rtwdemo_VectorOptimization

```

The optimized code is in `rtwdemo_VectorOptimization.c`. The signals `tmp1` and `tmp2` are the local variables `rtb_tmp1` and `rtb_tmp2`.

```

cfile = fullfile(cgDir, 'rtwdemo_VectorOptimization_grt_rtw', ...
    'rtwdemo_VectorOptimization.c');
rtwdemodbtype(cfile, '/* Model step', '/* Model initialize', 1, 0);

/* Model step function */
void rtwdemo_VectorOptimization_step(void)
{
    int32_T i;
    real_T rtb_tmp2;
    real_T rtb_tmp1;
    real_T rtb_Sum3;
    for (i = 0; i < 10; i++) {
        /* Gain: '<Root>/G2' incorporates:
         * UnitDelay: '<Root>/X2'
         */
        rtb_tmp2 = 0.3 * rtwdemo_VectorOptimization_DW.X2_DSTATE[i];

        /* Gain: '<Root>/G1' incorporates:
         * UnitDelay: '<Root>/X1'
         */
        rtb_tmp1 = 0.2 * rtwdemo_VectorOptimization_DW.X1_DSTATE[i];

        /* Sum: '<Root>/Sum3' incorporates:
         * Gain: '<Root>/G3'
         * Inport: '<Root>/In2'
         * Sum: '<Root>/Sum1'
         * Sum: '<Root>/Sum2'
         * UnitDelay: '<Root>/X3'
         */
        rtb_Sum3 = ((rtwdemo_VectorOptimization_U.In2[i] - 0.4 *
            rtwdemo_VectorOptimization_DW.X3_DSTATE[i]) - rtb_tmp2) -
            rtb_tmp1;

        /* Outport: '<Root>/Out2' */
        rtwdemo_VectorOptimization_Y.Out2[i] = rtb_Sum3;

        /* Update for UnitDelay: '<Root>/X3' */
        rtwdemo_VectorOptimization_DW.X3_DSTATE[i] = rtb_tmp2;
    }
}

```

```
/* Update for UnitDelay: '<Root>/X2' */
rtwdemo_VectorOptimization_DW.X2_DSTATE[i] = rtb_tmp1;

/* Update for UnitDelay: '<Root>/X1' */
rtwdemo_VectorOptimization_DW.X1_DSTATE[i] = rtb_Sum3;
}
}
```

Close the model and code generation report.

```
bdclose(model)
rtwdemoclean;
cd(currentDir)
```

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Minimize Computations and Storage for Intermediate Results at Block Outputs” on page 31-40
- “Use memcpy Function to Optimize Generated Code for Vector Assignments” on page 31-56

Enable and Reuse Local Block Outputs in Generated Code

In this section...

“Example Model” on page 31-102

“Generate Code Without Optimization” on page 31-103

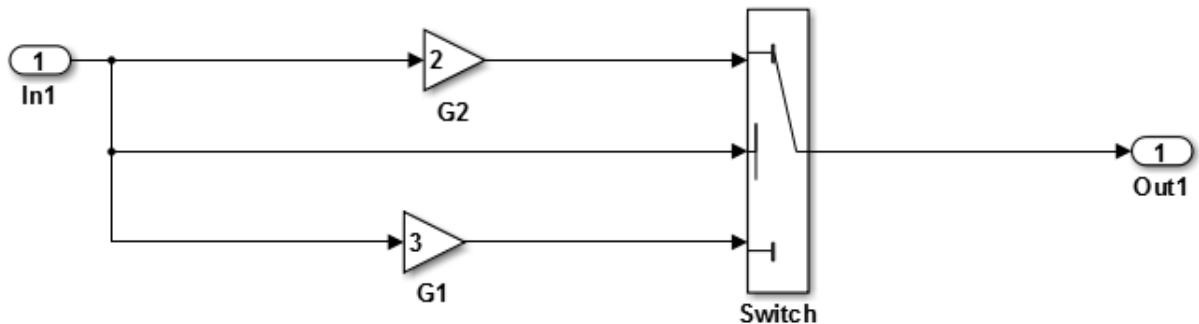
“Enable Local Block Outputs and Generate Code” on page 31-103

“Reuse Local Block Outputs and Generate Code” on page 31-104

This example shows how to specify block output as local variables. The code generator can potentially reuse these local variables in the generated code. Declaring block output as local variables conserves ROM consumption. Reusing local variables conserves RAM consumption and increases execution speed.

Example Model

- 1 Use Inport, Output, Gain, and Switch blocks to create the following model. In this example, the model is named `local_variable_ex`.



- 2 For G2, open the Gain Block Parameters dialog box. Enter a value of 2.
- 3 For G1, enter a value of 3.
- 4 For the Switch block, open the Block Parameters dialog box. For the **Criteria for passing first input** parameter, select `u2>=Threshold`.

Generate Code Without Optimization

- 1 Open the Model Configuration Parameters dialog box. Select the **Solver** pane. For the **Type** parameter, select **Fixed-step**.
- 2 Select the **All Parameters** tab and clear **Signal Storage Reuse**.
- 3 Select the **Code Generation > Report** pane and select **Create code generation report**.
- 4 Select the **Code Generation** pane. Select **Generate code only**, and then, in the model window, press **Ctrl+B**. When code generation is complete, an HTML code generation report appears.
- 5 In the code generation report, select the `local_variable_ex.c` section and view the model step function. The Gain block outputs are the global variables `local_variable_ex_B.G2` and `local_variable_ex_B.G1`.

```

/* Model step function */
void local_variable_ex_step(void)
{
    /* Switch: '<Root>/Switch' incorporates:
     * Inport: '<Root>/In1'
     */
    if (local_variable_ex_U.In1 >= 0.0) {
        /* Gain: '<Root>/G2' */
        local_variable_ex_B.G2 = 2.0 * local_variable_ex_U.In1;

        /* Output: '<Root>/Out1' */
        local_variable_ex_Y.Out1 = local_variable_ex_B.G2;
    } else {
        /* Gain: '<Root>/G1' */
        local_variable_ex_B.G1 = 3.0 * local_variable_ex_U.In1;

        /* Output: '<Root>/Out1' */
        local_variable_ex_Y.Out1 = local_variable_ex_B.G1;
    }

    /* End of Switch: '<Root>/Switch' */
}

```

Enable Local Block Outputs and Generate Code

- 1 Open the Model Configuration Parameters dialog box. On the **All Parameters** tab, select **Signal Storage Reuse**. **Signal Storage Reuse** enables the following optimization parameters:

- **Enable local block outputs**
 - **Reuse local block outputs**
 - **Eliminate superfluous local variables (expression folding)**
- 2 Clear **Reuse local block outputs** and **Eliminate superfluous local variables (expression folding)**.
 - 3 Generate code and view the model step function. There are three local variables in the model step function because you selected the optimization parameter **Enable Local Block Outputs**. The local variables `rtb_G2` and `rtb_G1` hold the outputs of the Gain blocks. The local variable `rtb_Switch` holds the output of the Switch block.

```
/* Model step function */
void local_variable_ex_step(void)
{
    real_T rtb_Switch;
    real_T rtb_G2;
    real_T rtb_G1;

    /* Switch: '<Root>/Switch' incorporates:
     * Inport: '<Root>/In1'
     */
    if (local_variable_ex_U.In1 >= 0.0) {
        /* Gain: '<Root>/G2' */
        rtb_G2 = 2.0 * local_variable_ex_U.In1;
        rtb_Switch = rtb_G2;
    } else {
        /* Gain: '<Root>/G1' */
        rtb_G1 = 3.0 * local_variable_ex_U.In1;
        rtb_Switch = rtb_G1;
    }

    /* End of Switch: '<Root>/Switch' */

    /* Output: '<Root>/Out1' */
    local_variable_ex_Y.Out1 = rtb_Switch;
}
```

Reuse Local Block Outputs and Generate Code

- 1 Open the Model Configuration Parameters dialog box. On the **Optimization > Signals and Parameters** pane, select **Reuse local block outputs**.

- 2 Generate code. In the `local_variable_ex.c` section, view the model step function. There is one local variable, `rtb_G2`, that the code generator uses three times.

```
/* Model step function */
void local_variable_ex_step(void)
{
    real_T rtb_G2;

    /* Switch: '<Root>/Switch' incorporates:
     * Inport: '<Root>/In1'
     */
    if (local_variable_ex_U.In1 >= 0.0) {
        /* Gain: '<Root>/G2' */
        rtb_G2 = 2.0 * local_variable_ex_U.In1;
    } else {
        /* Gain: '<Root>/G1' */
        rtb_G2 = 3.0 * local_variable_ex_U.In1;
    }

    /* End of Switch: '<Root>/Switch' */

    /* Outputport: '<Root>/Out1' */
    local_variable_ex_Y.Out1 = rtb_G2;
}
```

See Also

“Enable local block outputs”

Related Examples

- “Optimization Tools and Techniques” on page 31-13
- “Customize Stack Space Allocation” on page 31-93
- “Signal Representation in Generated Code” on page 14-109

Verification

Simulation and Code Comparison in Simulink Coder

Simulation and Code Comparison

In this section...

“Configure Signal Data for Logging” on page 32-2

“Log Simulation Data” on page 32-3

“Run Executable and Load Data” on page 32-5

“Visualize and Compare Results” on page 32-6

“Compare States for Simulation and Code Generation” on page 32-8

This example shows how to verify the answers computed by code generated from the `slexAircraftExample` model. It shows how to capture and compare two sets of output data. Simulating the model produces one set of output data. Executing the generated code produces a second set of output data.

Note To obtain a valid comparison between model output and the generated code, use the same **Solver options** and **Step size** for the simulation run and the build process.

Configure Signal Data for Logging

Configure the model for logging and recording signal data.

- 1 Make sure that `slexAircraftExample` is closed. Clear the base workspace to eliminate the results of previous simulation runs. In the Command Window, type:

```
clear
```

The clear operation clears variables created during previous simulations and all workspace variables, some of which are standard variables that the `slexAircraftExample` model requires.

- 2 In the Command Window, enter `slexAircraftExample` to open the model.
- 3 In the model window, choose **File > Save As**, navigate to the working folder, and save a copy of the `slexAircraftExample` model as `myAircraftExample`.
- 4 Set up your model to log signal data for signals: `Stick`, `alpha`, `rad`, and `q`, `rad/sec`. For each signal:
 - a Right-click the signal. From the context menu, select **Properties**.

- b** In the Signal Properties dialog box, select **Log signal data**.
- c** In the **Logging name** section, from the drop-down list, select **Custom**.
- d** In the text field, enter the logging name for the corresponding signal.

Signal Name	Logging Name
Stick	Stick_input
alpha, rad	Alpha
q, rad/sec	Pitch_rate

- e** Click **Apply** and **OK**.

For more information, see “Export Signal Data Using Signal Logging”.

- 5** Select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box.
- 6** Select the **Solver** pane and in the **Solver options** section, specify the **Type** parameter as **Fixed-step**.
- 7** On the **Data Import/Export** pane:
 - Specify the **Format** parameter as **Structure with time**.
 - Clear the **States** parameter check box.
 - Select the **Signal logging** parameter.
 - Select the **Record logged workspace data in Simulation Data Inspector** parameter to enable logged signal data to send to the Simulation Data Inspector after the simulation is finished.
- 8** Save the model.

Proceed to “Log Simulation Data” on page 32-3.

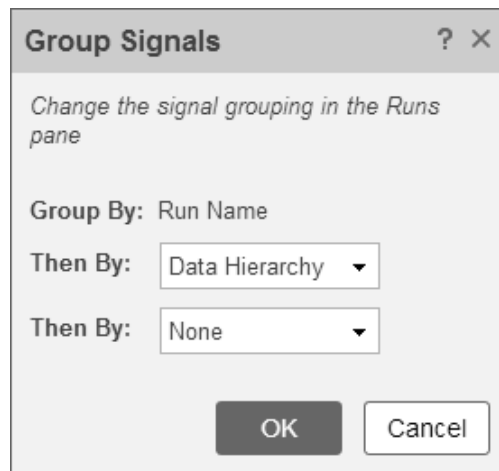
Log Simulation Data

Run the simulation, log the signal data, and view the data in the Simulation Data Inspector.

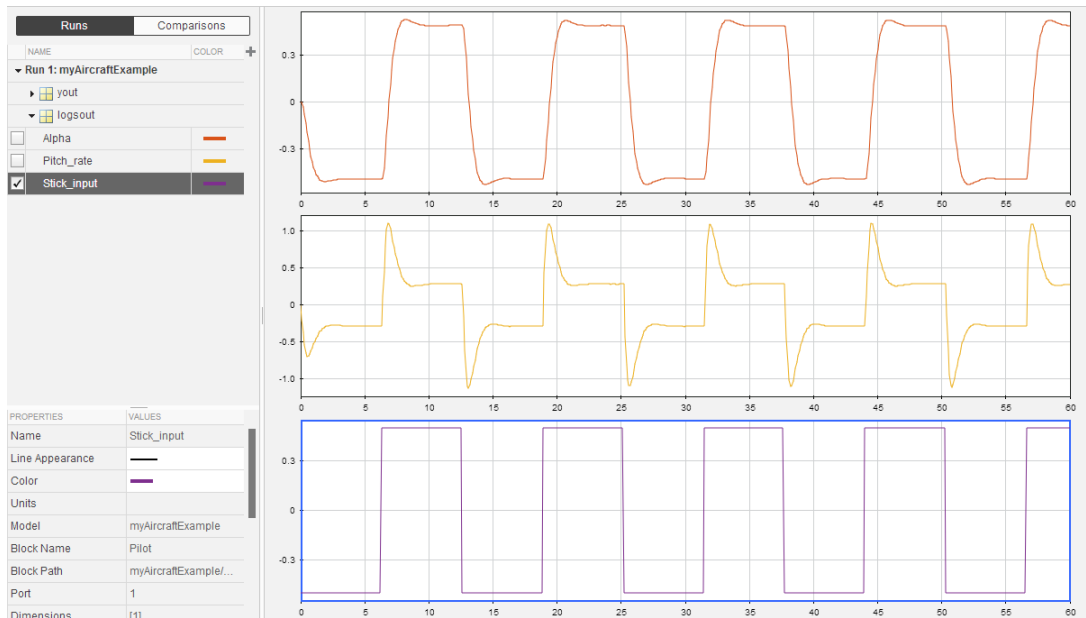
- 1** Run the model. When the simulation is done, on the Simulink Editor toolbar, the **Simulation Data Inspector** button is highlighted to indicate that new simulation output is available in the Simulation Data Inspector.



- 2 Click the **Simulation Data Inspector** button to open the Simulation Data Inspector.
- 3 Group the signals:
 - a On the **Visualize** tab, click **Group Signals**.
 - b In the Group Signals dialog box, select **Data Hierarchy** from the **Then By** list.



- c Click **OK**.
- 4 Click the **logout** expander to view the logged signals.
- 5 Click the **Format** tab.
- 6 Click the **Subplots** button and select **3x1** to show three subplots.
- 7 For each signal:
 - a Click the top subplot. A blue border indicates the plot selection.
 - b Select the check box next to the **Alpha** signal name. The signal data appears in the subplot.
 - c Plot the **Pitch_rate** signal in the middle subplot.
 - d Plot the **Stick_input** signal in the bottom subplot.



Proceed to “Run Executable and Load Data” on page 32-5.

Run Executable and Load Data

You must rebuild and run the `myAircraftExample` executable to obtain a valid data file because you have modified the model.

- 1 Select **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box.
- 2 Select the **Code Generation > Interface** pane.
- 3 Set the **MAT-file variable name modifier** parameter to `rt_`. `rt_` is prefixed to each variable that you selected for logging in the first part of this example.
- 4 Click **Apply** and **OK**.
- 5 Save the model.
- 6 On the Simulink Editor toolbar, click the **Build Model** button to generate code.
- 7 When the build is finished, run the standalone program from the Command Window.

```
!myAircraftExample
```

The executing program writes the following messages to the Command Window.

```
** starting the model **  
** created myAircraftExample.mat **
```

8 Load the data file `myAircraftExample.mat`.

```
load myAircraftExample
```

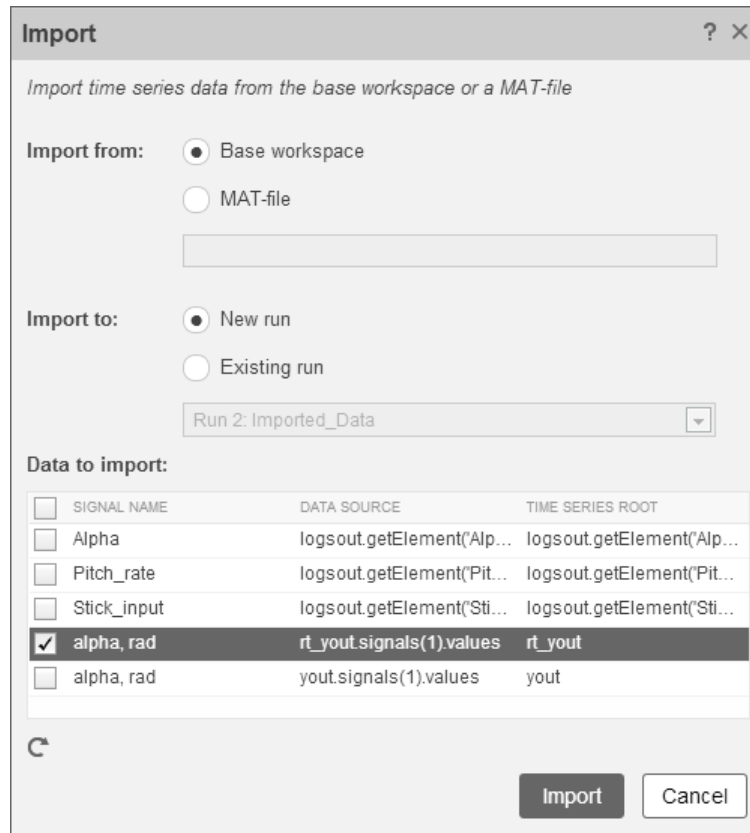
Tip For UNIX platforms, run the executable in the Command Window with the syntax `!./executable_name`. If preferred, run the executable from an OS shell with the syntax `./executable_name`. For more information, see “Run External Commands, Scripts, and Programs”.

Proceed to “Visualize and Compare Results” on page 32-6.

Visualize and Compare Results

When you follow the example sequence that began in “Configure Signal Data for Logging” on page 32-2, you obtain data from a Simulink run of the model and from a run of the program generated from the model.

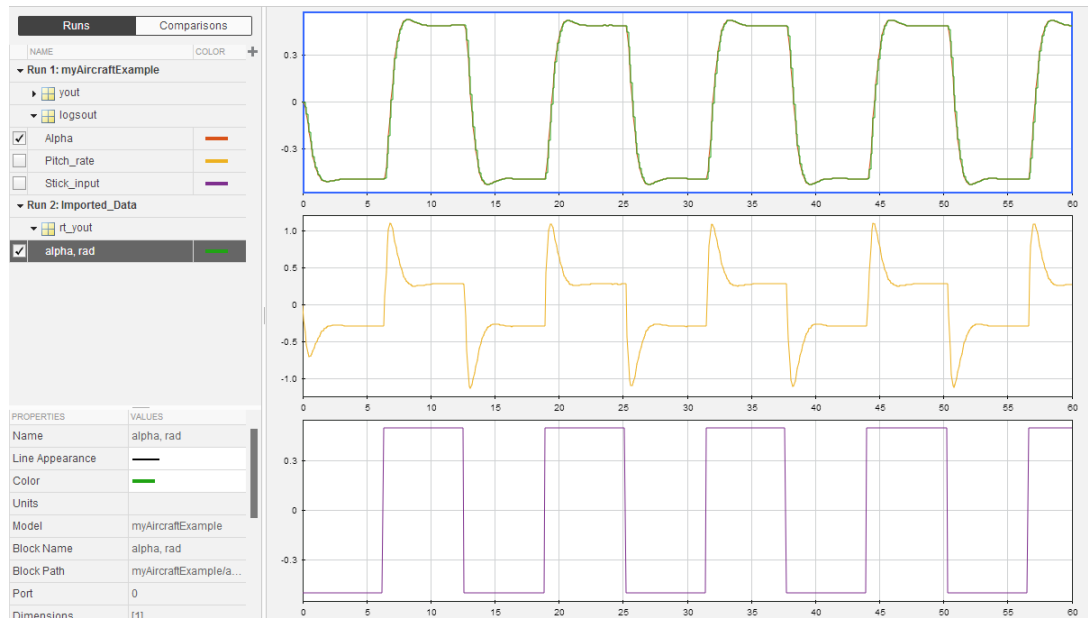
- 1** To view the execution output for `alpha, rad`, import the data into the Simulation Data Inspector.
 - a** On the Simulation Data Inspector **Visualize** tab, click the **Import** button to open the Import dialog.
 - b** Specify **Import from** as **Base workspace**.
 - c** Specify **Import to** as **New run**.
 - d** To the left of **Signal Name**, click the check mark to clear the check boxes.
 - e** Select the check box for the `alpha, rad` data where the **Time Series Root** is `rt_yout`.
 - f** Click **Import**.



The selected data is now under **Run 2: Imported_Data**.

- 2 View a plot of the executed data.
 - a Click the `rt_yout` expander.
 - b Click the top subplot and select the check box next to the `alpha, rad` signal name. The signal data appears in the top subplot.

The `alpha, rad` signal from Run 1 and Run 2 overlap in the subplot because the signals are equivalent.



It is possible to see a very small difference between simulation and code generation results. A slight difference can be caused by many factors, including:

- Different compiler optimizations
- Statement ordering
- Run-time libraries

For example, a function call such as `sin(2.0)` can return a slightly different value depending on which C library you use. Such variations can also cause differences between your results and these results.

Compare States for Simulation and Code Generation

The order in which Simulink logs states during simulation is different than the order in which Simulink Coder logs states during code generation. If you want to compare states between simulation and code generation, sort the states by block name.

For example, by default, Simulink exports state data to the MATLAB variable, `xout`. Simulink Coder exports state data to the variable `rt_xout`. To sort the state data for these variables, enter the following commands in the MATLAB Command Window:

```
[~,idx1]=sort({xout.signals.blockName});  
xout_sorted=[xout.signals(idx1).values];  
[~,idx2]=sort({rt_xout.signals.blockName});  
rt_xout_sorted=[rt_xout.signals(idx2).values];
```

You can confirm that the logging order is the same between code generation and simulation by entering the following command in the MATLAB Command Window:

```
isequal(xout_sorted, rt_xout_sorted)
```

Related Examples

- “Log Program Execution Results” on page 27-2

Numerical Consistency between Model and Generated Code

Numerical Consistency of Model and Generated Code Simulation Results

In this section...

“Numerical Consistency” on page 33-2

“Numerical Consistency in Complex Systems” on page 33-3

“Reasons for Block-Level Numerical Differences” on page 33-5

Numerical Consistency

In the Model-Based Design workflow, you use MathWorks products to generate code for numerical applications that employ fixed-point and floating-point arithmetic.

- To develop models, you use MATLAB, Simulink, and Stateflow.
- To generate source code, you use Simulink Coder and Embedded Coder.
- To test numerical equivalence between your model and generated code, you compare model and generated code simulation results. For example, normal mode simulation results compared with software-in-the-loop (SIL) simulation results.

The results from the model and generated code simulations are numerically consistent if:

- In fixed-point applications, the results agree in a bit-wise comparison.
- In floating-point applications, the results agree with an error tolerance that you specify.

Use the Simulation Data Inspector to compare results. To determine whether discrepancies exist or are significant, you can specify absolute and relative tolerance values:

- For fixed-point applications, you can specify an absolute tolerance of zero.
- For floating-point applications, you can specify tolerance with respect to a reference value or signal. The choice of reference depends on your application. Consider these examples:
 - An algorithm that solves a linear algebraic equation by iterative, feed-forward error calculations. You can specify tolerance with respect to `eps`.

- A Proportional-Integral-Derivative (PID) controller for a closed-loop system. For transient behavior, you can specify tolerance with criteria from a standard. For steady-state behavior, you can specify tolerance with reference to the PID controller characteristics.

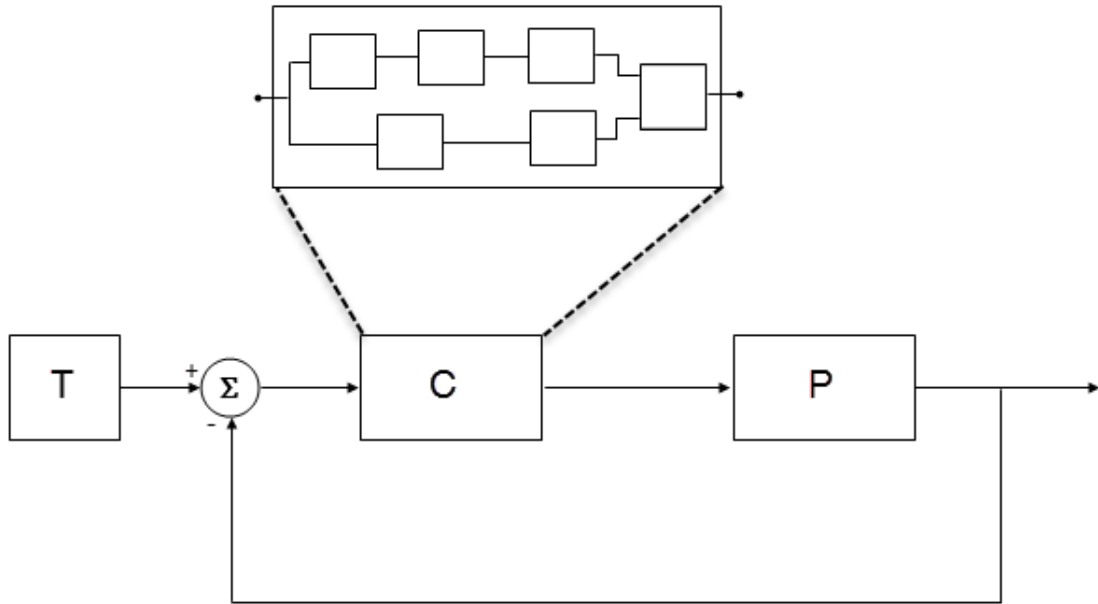
Programmatically, you can specify absolute and relative tolerance values through the `absTol` and `relTol` properties of the `Simulink.sdi.Signal` object.

Numerical Consistency in Complex Systems

For complex systems, numerical differences between model and generated code simulations can be a result of block-level differences propagating through the system. If you observe numerical differences at the system level:

- 1 Identify blocks for which block-level numerical differences exceed the error tolerance.
- 2 Investigate each identified block.

Consider the following plant-controller model.



- T produces reference or test signals.
- C is the controller component. The controller output is the plant input. C can be a Model block that comprises multiple Model blocks.
- P is the plant component. The plant output is subtracted from the reference signal to produce the controller input.

To test numerical equivalence between the model controller and the generated code version:

- 1 Run the model in normal mode, and, using the Simulation Data Inspector, record the output of C.
- 2 Specify SIL mode for C. Rerun the simulation, recording the output of C.
- 3 Using the Simulation Data Inspector, compare normal and SIL mode outputs with reference to your specified error tolerance.

If the Simulation Data Inspector comparison indicates a match, the model and generated code results are numerically consistent.

If the normal and SIL mode outputs do not match:

- 1 Within C, enable signal logging for block outputs.
- 2 Run the model in normal mode.
- 3 Rerun the simulation with C in SIL mode.
- 4 Using the Simulation Data Inspector, compare the logged output signals with reference to your specified error tolerance. See “Compare Signal Data from Multiple Simulations”.
- 5 Identify blocks for which normal and SIL mode output differences exceed the error tolerance.
- 6 Analyze each identified block and look for the cause. For example, the generated code might use a different math library than MATLAB.

Note: If the comparison of a large number of signals is required, you can automate the workflow with Simulink Test™. See “Code Generation Verification Workflow with Simulink Test”.

Reasons for Block-Level Numerical Differences

In fixed-point and floating-point application development, there are factors that can affect numerical agreement between block-level results from model and generated code simulations.

Some factors can affect both fixed-point and floating-point applications. For example, the use of:

- Code generation optimization.
- Custom code.
- Code replacement library entries whose results differ from MATLAB results.
- Code replacement libraries that implement different algorithms.

Other factors affect only floating-point applications. For example:

- Numerical soundness of algorithm.

- Algorithm sensitivity to input.
- Closed loop and open loop behavior.

References

- [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

See Also

eps

Related Examples

- “Record Logged Simulation Data”
- “Compare Signal Data from Multiple Simulations”
- “Inspect and Compare Signal Data Programmatically”
- “Code Generation Verification Workflow with Simulink Test”
- “Configure Code Optimizations”

More About

- “How the Simulation Data Inspector Compares Time Series Data”
- “Differences in Behavior After Compiling MATLAB Code”
- “Code Replacement”
- “Types of In-the-Loop Testing in the V-Model”
- MATLAB Function
- “SIL and PIL Limitations”

Supported Hardware

Embedded Targets

Simulink Coder Supported Hardware

As of this release, Simulink Coder supports the following hardware.

Support Package	Vendor	Earliest Release Available	Last Release Available
ARM Cortex-Based VEX Microcontroller	VEX [®] Robotics	R2016a	Current
NXP FRDM-K64F Board	NXP [™]	R2016b	Current
NXP FRDM-KL25Z Board	NXP	R2016b	Current
STMicroelectronics Nucleo boards	STMicroelectr	R2016b	Current

For a complete list of supported hardware, see [Hardware Support](#).

Customization

Build Process Integration in Simulink Coder

- “Control Build Process Compiling and Linking” on page 35-2
- “Cross-Compile Code Generated on Microsoft Windows” on page 35-4
- “Control Library Location and Naming During Build” on page 35-7
- “Recompile Precompiled Libraries” on page 35-13
- “Customize Post-Code-Generation Build Processing” on page 35-14
- “Configure Generated Code with TLC” on page 35-22
- “Use `rtwmakecfg.m` API to Customize Generated Makefiles” on page 35-24
- “Customize Build Process with `STF_make_rtw_hook` File” on page 35-29
- “Customize Build Process with `sl_customization.m`” on page 35-36
- “Replace `STF_rtw_info_hook` Supplied Target Data” on page 35-41
- “Customize Build to Use Shared Utility Code” on page 35-42

Control Build Process Compiling and Linking

After generating code for a model, the build process determines whether or not to compile and link an executable program. This decision is governed by the following:

- **Generate code only** option

When you select this option, the code generator produces code for the model, including a makefile.

- **Generate makefile** option

When you clear this option, the code generator does not produce a makefile for the model. You must specify post code generation processing, including compilation and linking, as a user-defined command, as explained in “Customize Post-Code-Generation Build Processing” on page 35-14.

- **Makefile-only target**

The Microsoft Visual C++ Project Makefile versions of the `grt` and Embedded Coder target configurations generate a Visual C++ project makefile (`model.mak`). To build an executable, you must open `model.mak` in the Visual C++ IDE and compile and link the model code.

- **HOST template makefile variable**

The template makefile variable `HOST` identifies the type of system upon which your executable is intended to run. The variable can be set to one of three possible values: `PC`, `UNIX`, or `ANY`.

By default, `HOST` is set to `UNIX` in template makefiles designed for use with The Open Group UNIX platforms (such as `grt_unix.tmf`), and to `PC` in the template makefiles designed for use with development systems for the PC (such as `grt_vc.tmf`).

If the Simulink software is running on the same type of system as that specified by the `HOST` variable, then the executable is built. Otherwise,

- If `HOST = ANY`, an executable is still built. This option is useful when you want to cross-compile a program for a system other than the one the Simulink software is running on.
- Otherwise, processing stops after generating the model code and the makefile; the following message is displayed on the MATLAB command line.

```
### Make will not be invoked - template makefile is for a different host
```

- TGT_FCN_LIB template makefile variable

The template makefile variable TGT_FCN_LIB specifies compiler command line options. The line in the makefile is TGT_FCN_LIB = |>TGT_FCN_LIB<|. Use this token in a makefile conditional statement to specify a standard math library as a compiler option. Possible |>TGT_FCN_LIB<| token values are:

Value	Generates Calls To
Name of custom CRL	ISO®/IEC 9899:1990 C (ANSI_C) standard math library
ISO_C	ISO/IEC 9899:1999 C standard math library
ISO_C++	ISO/IEC 14882:2003 C++ standard math library
GNU	GNU extensions to the ISO/IEC 9899:1999 C standard math library

Cross-Compile Code Generated on Microsoft Windows

If you need to generate code with the code generator on a Microsoft Windows system but compile the generated code on a different supported platform, you can do so by modifying your TMF and model configuration parameters. For example, you would need to do this if you develop applications with the MATLAB and Simulink products on a Windows system, but you run your generated code on a Linux system.

To set up a cross-compilation development environment, do the following (here a Linux system is the destination platform):

- 1 On your Windows system, copy the UNIX TMF for your target to a local folder. This will be your working folder for initiating code generation. For example, you might copy the file `matlabroot/rtw/c/grt/grt_unix.tmf` to `D:/work/my_grt_unix.tmf`.
- 2 Make the following changes to your copy of the TMF:
 - Add the following line near the `SYS_TARGET_FILE =` line:

```
MAKEFILE_FILESEP = /
```
 - Search for the line `'ifeq ($(OPT_OPTS),$(DEFAULT_OPT_OPTS))'` and, for each occurrence, remove the conditional logic and retain only the `'else'` code. That is, remove everything from the `'if'` to the `'else'`, inclusive, as well as the closing `'endif'`. Only the lines from the `'else'` portion should remain. This forces the run-time libraries to build for a Linux system.
- 3 Open your model and make the following changes in the **Code Generation** pane of the Configuration Parameters dialog:
 - Specify the name of your new TMF in the “Template makefile” text box (for example, `my_grt_unix.tmf`).
 - Select **Generate code only** and click **Apply**.
- 4 Generate the code.
- 5 If the build folder (folder from which the model was built) is not already Linux accessible, copy it to a Linux accessible path. For example, if your build folder for the generated code was `D:\work\mymodel_grt_rtw`, copy that entire folder tree to a path such as `/home/user/mymodel_grt_rtw`.
- 6 If the MATLAB folder tree on the Windows system is Linux accessible, skip this step. Otherwise, copy the include and source folders to a Linux accessible drive partition, for example, `/home/user/myinstall`. These folders appear in the makefile after

MATLAB_INCLUDES = and ADD_INCLUDES = and can be found by searching for \$(MATLAB_ROOT). Paths that contain \$(MATLAB_ROOT) must be copied. Here is an example list (your list will vary depending on your model):

```
$(MATLAB_ROOT)/rtw/c/grt
$(MATLAB_ROOT)/extern/include
$(MATLAB_ROOT)/simulink/include
$(MATLAB_ROOT)/rtw/c/src
$(MATLAB_ROOT)/rtw/c/tools
```

Additionally, paths containing \$(MATLAB_ROOT) in the build rules (lines with %.o :) must be copied. For example, based on the build rule

```
%.o : $(MATLAB_ROOT)/rtw/c/src/ext_mode/tcpip/%.c
the following folder should be copied:
```

```
$(MATLAB_ROOT)/rtw/c/src/ext_mode/tcpip
```

Note: The path hierarchy relative to the MATLAB root must be maintained. For example, c:\MATLAB\rtw\c\tools* would be copied to /home/user/mlroot/rtw/c/tools/*.

For some blocksets, it is easiest to copy a higher-level folder that includes the subfolders listed in the makefile. For example, the DSP System Toolbox product requires the following folders to be copied:

```
$(MATLAB_ROOT)/toolbox/dspblks
$(MATLAB_ROOT)/toolbox/rtw/dspblks
```

7 Make the following changes to the generated makefile:

- Set both MATLAB_ROOT and ALT_MATLAB_ROOT equal to the Linux accessible path to *matlabroot* (for example, home/user/myinstall).
- Set COMPUTER to the computer value for your platform, such as GLNX86. Enter `help computer` in the MATLAB Command Window for a list of computer values.
- In the ADD_INCLUDES list, change the build folder (designating the location of the generated code on the Windows system) and parent folders to Linux accessible include folders. For example, change D:\work\mymodel_grt_rtw\ to /home/user/mymodel_grt_rtw.

Additionally, if *matlabroot* is a UNC path, such as \\my-server\myapps\matlab, replace the hard-coded MATLAB root with \$(MATLAB_ROOT).

- 8 From a Linux shell, compile the code you generated on the Windows system. You can do this by running the generated *model.bat* file or by typing the make command line as it appears in the *.bat* file.

Note: If errors occur during makefile execution, you may need to run the `dos2unix` utility on the makefile (for example, `dos2unix mymodel.mk`).

Related Examples

- Use packNGo to Relocate Code to Another Development Environment

Control Library Location and Naming During Build

Use the `TargetPreCompLibLocation` and `TargetLibSuffix` configuration parameters to control values in generated makefiles during model builds when you use the toolchain approach or the template makefile approach.

In this section...

“Library Control Parameters” on page 35-7

“Specify the Location of Precompiled Libraries” on page 35-9

“Control the Location of Model Reference Libraries” on page 35-10

“Control the Suffix Applied to Library File Names” on page 35-11

Library Control Parameters

Use the library control parameters to:

- Specify the location of precompiled libraries, such as blockset libraries or the Simulink Coder block library. Typically, a target has cross-compiled versions of these libraries and places them in a target-specific folder.
- Control the suffix applied to library file names (for example, `_target.a` or `_target.lib`).

Targets can set the parameters inside the system target file (STF) select callback. For example:

```
function mytarget_select_callback_handler(varargin)
    hDig=varargin{1};
    hSrc=varargin{2};
    slConfigUISetVal(hDig, hSrc, 'TargetPreCompLibLocation',...
        'c:\mytarget\precomplibs');
    slConfigUISetVal(hDig, hSrc, 'TargetLibSuffix',...
        '_target.library');
```

The TMF has corresponding expansion tokens:

```
|>EXPAND_LIBRARY_LOCATION<|
|>EXPAND_LIBRARY_SUFFIX<|
```

Alternatively, you can use a call to the `set_param` function. For example:

```
set_param(model, 'TargetPreCompLibLocation', ...
```

```
'c:\mytarget\precomplib');
```

Note: If your model contains referenced models, you can use the make option `USE_MDLREF_LIBPATHS` to control whether libraries used by the referenced models are copied to the parent model's build folder. For more information, see “Control the Location of Model Reference Libraries” on page 35-10.

Using TargetLibSuffix with the Toolchain Approach

With `TargetLibSuffix`, you specify a suffix followed by an extension. For example:
suffix.extension

However, when you use the toolchain approach, only the suffix provided by `TargetLibSuffix` is honored. The extension that the `TargetLibSuffix` provides is not honored because the toolchain approach provides a different extension.

For example, with the target makefile approach, the final binary name is composed of the *modelName*, the *suffix*, and the *extension* provided by `TargetLibSuffix`:

```
modelName+suffix.extension_from_TargetLibSuffix
```

With the toolchain approach, the final binary name is composed of the *modelName*, the *suffix*, and the *extension* provided by the toolchain approach:

```
model+suffix.extension_from_toolchain_approach
```

The extension that the toolchain approach uses comes from the file extension of the static library that the build tool creates. To get this information:

Create the toolchain object. For example, enter:

```
tc = ToolchainInfo used in the model
```

Get the build tool name. For example, enter:

```
tool = tc.getBuildTool('C Compiler');  
or
```

```
tool = tc.getBuildTool('C++ Compiler');
```

Get the extension. For example, enter:

```
extension_from_ToolchainInfo = tool.getFileExtension('Static Library')
```

Note: Note: If you do not set the `TargetLibSuffix` parameter, template makefile and toolchain approaches behave identically. See “Customize Library File Suffix and File Type” on page 8-27.

Specify the Location of Precompiled Libraries

Use the `TargetPreCompLibLocation` configuration parameter to:

- Override the precompiled library location specified in the `rtwmakecfg.m` file (see “Use `rtwmakecfg.m` API to Customize Generated Makefiles” on page 35-24 for details)
- Precompile and distribute target-specific versions of product libraries (for example, the DSP System Toolbox product)

For a precompiled library, such as a blockset library or the Simulink Coder block library, the location specified in `rtwmakecfg.m` is typically a location specific to the blockset or the Simulink Coder product. The code generator expects that the library exists in this location and links against the library during builds.

However, for some applications, such as custom targets, it is preferable to locate the precompiled libraries in a target-specific or other alternate location rather than in the location specified in `rtwmakecfg.m`. For a custom target, the code generator expects that the target-specific cross-compiler creates the library, and you place the library in the target-specific location. Libraries supported by the target should be compiled and placed in the target-specific location so they can be used during the build process.

You can set up the `TargetPreCompLibLocation` parameter in its select callback. The path that you specify for the parameter must be a fully qualified, absolute path to the library location. Relative paths are not supported. For example:

```
slConfigUISetVal(hDlg, hSrc, 'TargetPreCompLibLocation',...
'c:\mytarget\precomplibs');
```

Alternatively, you set the parameter with a call to the `set_param` function. For example:

```
set_param(model, 'TargetPreCompLibLocation',...
'c:\mytarget\precomplibs');
```

During makefile generation, the build process replaces the tokens with the location from the `rtwmakecfg.m` file. For example, if the library name in the `rtwmakecfg.m` file is `'rtwlib'`, the template makefile build approach expands the token from:

```
LIBS += |>EXPAND_LIBRARY_LOCATION<|>EXPAND_LIBRARY_NAME<|>
```

```
_target.library
```

to:

```
LIBS += c:\mytarget\precomplibs\rtwlib_target.library
```

By default, `TargetPreCompLibLocation` is an empty character vector. The build process uses the location in `rtwmakecfg.m` for the token replacement.

Control the Location of Model Reference Libraries

On platforms other than the Apple Macintosh platform, when building a model that uses referenced models, the default build process includes:

- Copy libraries that the referenced models uses to the parent model's build folder.
- Assign the file names of the libraries to `MODELREF_LINK_LIBS` in the generated makefile.

For example, if a model includes a referenced model `sub`, the build process assigns the library name `sub_rtwlib.lib` to `MODELREF_LINK_LIBS`. The build process copies the library file to the parent model's build folder. This definition is then used in the final link line, which links the library into the final product (usually an executable). This technique minimizes the length of the link line.

On the Macintosh platform, and optionally on other platforms, the build process includes:

- No copying of libraries that the referenced models uses to the parent model's build folder.
- Assign the relative paths and file names of the libraries to `MODELREF_LINK_LIBS` in the generated makefile.

When using this technique, the build process assigns a relative path such as `../slprj/grt/sub/sub_rtwlib.lib` to `MODELREF_LINK_LIBS`. The build process uses the path to gain access to the library file at link time.

To change to the nondefault behavior on platforms other than the Macintosh platform, select the **Configuration Parameters > Code Generation > Make command** field. Enter:

```
make_rtw USE_MDLREF_LIBPATHS=1
```

If you specify other Make command arguments, such as `OPTS=" -g "`, the order in which you specify the multiple arguments does not matter.

To return to the default behavior, set `USE_MDLREF_LIBPATHS` to 0, or remove it.

Control the Suffix Applied to Library File Names

Use the `TargetLibSuffix` configuration parameter to control the suffix applied to library names (for example, `_target.lib` or `_target.a`). The specified suffix scheme must include a period (.). You can apply `TargetLibSuffix` to the following libraries:

- Libraries on which a target depends, as specified in the `rtwmakecfg.m` API. You can use `TargetLibSuffix` to change the suffix of both precompiled and non-precompiled libraries configured from the `rtwmakecfg` API. For details, see “Use `rtwmakecfg.m` API to Customize Generated Makefiles” on page 35-24.

In this case, a target can set the parameter in its select callback. For example:

```
s1ConfigUISetVal(hDlg, hSrc, 'TargetLibSuffix',...
'_target.library');
```

Alternatively, you can use a call to the `set_param` function. For example:

```
set_param(model, 'TargetLibSuffix', '_target.library');
```

During the TMF-to-makefile conversion, the build process replaces the token `|>EXPAND_LIBRARY_SUFFIX<|` with the specified suffix. For example, if the library name specified in the `rtwmakecfg.m` file is `'rtwlib'`, the TMF expands from:

```
LIBS += |>EXPAND_LIBRARY_LOCATION<|\ |>EXPAND_LIBRARY_NAME<|\
|>EXPAND_LIBRARY_SUFFIX<|
```

to:

```
LIBS += c:\mytarget\precomplibs\rtwlib_target.library
```

By default, `TargetLibSuffix` is set to an empty character vector. In this case, the build process replaces the token `|>EXPAND_LIBRARY_SUFFIX<|` with an empty character vector.

- Shared utility library and the model libraries created with model reference. For these cases, associated makefile variables do not require the `|>EXPAND_LIBRARY_SUFFIX<|` token. Instead, the build process includes `TargetLibSuffix` implicitly. For example, for a top model named `topmodel` with referenced models named `refmodel1` and `refmodel2`, the top model's TMF is expanded from:

```
SHARED_LIB = |>SHARED_LIB<|
```

```
MODELLIB          = |>MODELLIB<|  
MODELREF_LINK_LIBS = |>MODELREF_LINK_LIBS<|
```

to:

```
SHARED_LIB        = \  
..\slprj\ert\_sharedutils\rtwshared_target.library  
MODELLIB          = topmodellib_target.library  
MODELREF_LINK_LIBS = \  
refmodel11_rtwlib_target.library refmodel12_rtwlib_target.library
```

By default, the `TargetLibSuffix` parameter is an empty character vector. In this case, the build process chooses a default suffix for these three tokens using a file extension of `.lib` on Windows hosts and `.a` on UNIX hosts. (For model reference libraries, the default suffix additionally includes `_rtwlib`.) For example, on a Windows host, the expanded makefile values are:

```
SHARED_LIB        = ..\slprj\ert\_sharedutils\rtwshared.lib  
MODELLIB          = topmodellib.lib  
MODELREF_LINK_LIBS = refmodel11_rtwlib.lib refmodel12_rtwlib.lib
```

Recompile Precompiled Libraries

You can recompile precompiled libraries included as part of the code generator, such as `rtwlib` or `dsplib`, by using a supplied MATLAB function, `rtw_precompile_libs`. You might consider doing this if you need to customize compiler settings for various platforms or environments. For details on using `rtw_precompile_libs`, see “Precompile S-Function Libraries” on page 24-162.

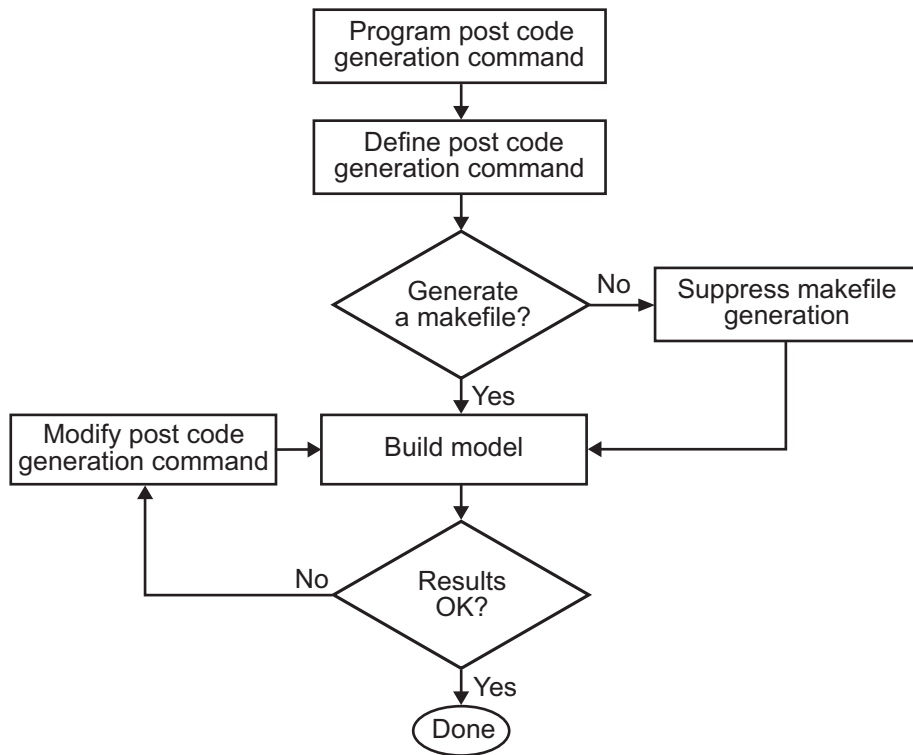
Customize Post-Code-Generation Build Processing

The code generator provides a set of tools, including a build information object, you can use to customize build processing that occurs after code generation. You might use such customizations for target development or the integration of third-party tools into your application development environment.

In this section...
“Workflow for Setting Up Customizations” on page 35-14
“Build Information Object” on page 35-15
“Program a Post Code Generation Command” on page 35-16
“Define a Post Code Generation Command” on page 35-17
“Customize Build Process with PostCodeGenCommand and Relocate Generated Code to an External Environment” on page 35-18
“Suppress Makefile Generation” on page 35-20

Workflow for Setting Up Customizations

The following figure and the steps that follow show the general workflow for setting up post-code-generation customizations.



- 1 Program the post code generation command.
- 2 Define the post code generation command.
- 3 Suppress makefile generation, if applicable.
- 4 Build the model.
- 5 Modify the command and rebuild the model until the build results are acceptable.

Build Information Object

At the start of a model build, the build process logs the following build option and dependency information to a temporary build information object:

- Compiler options
- Preprocessor identifier definitions
- Linker options

- Source files and paths
- Include files and paths
- Precompiled external libraries

You can retrieve information from and add information to this object by using an extensive set of functions. For a list of available functions and detailed function descriptions, see “Build Process Customization”. “Program a Post Code Generation Command” on page 35-16 explains how to use the functions to control post code generation build processing.

Program a Post Code Generation Command

For certain applications, you might want to control aspects of the build process after the code generation. For example, you might do this if you develop your own target, or you want to apply an analysis tool to the generated code before continuing with the build process. You can apply this level of control to the build process by programming and then defining a post code generation command.

A post code generation command is a MATLAB language file that typically calls functions that get data from or add data to the model's build information object. You can program the command as a script or function.

If You Program the Command as a...	Then the...
Script	Script can gain access to the model name and the build information directly
Function	Function can pass the model name and the build information as arguments

If your post code generation command calls user-defined functions, make sure the functions are on the MATLAB path. If the build process cannot find a function you use in your command, the build process errors out.

You can then call a combination of build information functions, as listed in “Build Process Customization”, to customize the model's post code generation build processing.

The following example shows a fragment of a post code generation command that gets the file names and paths of the source and include files generated for a model for analysis.

```
function analyzegeneratecode(buildInfo)
% Get the names and paths of source and include files
% generated for the model and then analyze them.

% buildInfo - build information for my model.

% Define cell array to hold data.
MyBuildInfo={};

% Get source file information.
MyBuildInfo.srcfiles=getSourceFiles(buildInfo, true, true);
MyBuildInfo.srcpaths=getSourcePaths(buildInfo, true);

% Get include (header) file information.
MyBuildInfo.incfiles=getIncludeFiles(buildInfo, true, true);
MyBuildInfo.incpaths=getIncludePaths(buildInfo, true);

% Analyze generated code.
.
.
.
```

Define a Post Code Generation Command

After you program a post code generation command, you need to inform the build process that the command exists and to add it to the model's build processing. You do this by defining the command with the `PostCodeGenCommand` model configuration parameter. When you define a post code generation command, the build process evaluates the command after generating and writing the model's code to disk and before generating a makefile.

As the following syntax lines show, the arguments that you specify when setting the configuration parameter varies depending on whether you program the command as a script, function, or set of functions.

Note: When defining the command as a function, you can specify an arbitrary number of input arguments. To pass the model's name and build information to the function, specify identifiers `modelName` and `buildInfo` as arguments.

Script

```
set_param(model, 'PostCodeGenCommand', ...
'pcgScriptName');
```

Function

```
set_param(model, 'PostCodeGenCommand',...  
    'pcgFunctionName(modelName)');
```

Multiple Functions

```
pcgFunctions=...  
'pcgFunction1Name(modelName);...  
pcgFunction2Name(buildInfo)';  
set_param(model, 'PostCodeGenCommand',...  
    pcgFunctions);
```

The following call to `set_param` defines `PostCodGenCommand` to evaluate the function `analyzezencode`.

```
set_param(model, 'PostCodeGenCommand',...  
    'analyzezencode(buildInfo)');
```

Customize Build Process with PostCodeGenCommand and Relocate Generated Code to an External Environment

This example shows how to use the Build Information API and the **Post Code Generation Command** parameter, `PostCodeGenCommand`.

The `PostCodeGenCommand` parameter value is `rtwdemo_buildinfo_data`. This value directs the build process to invoke the function after code generation.

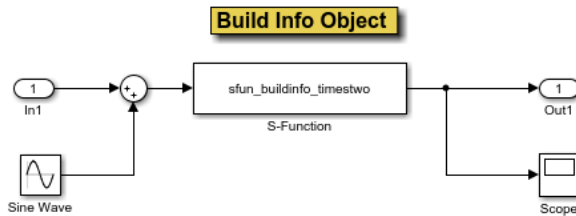
The example also demonstrates how to use the `rtwmakecfg.m` API.

For more information, click on the documentation links in the model.

Open Example Model

Open the example model `rtwdemo_buildinfo`.

```
open_system('rtwdemo_buildinfo');
```



Generate Code Using Simulink Coder (double-click)

Generate Code Using Embedded Coder (double-click)

[Open rtwmakecfg.m](#)
[Open rtwdemo_buildinfo_data.m](#)

Open BuildInfo.html

Description
 This example shows how to use the Build Information API as well as the Post Code Generation Command parameter, PostCodeGenCommand. For more information, click on the documentation links below.
 The function rtwdemo_buildinfo_data is invoked during the build process. Additionally, the rtwmakecfg.m API is shown.

Instructions

1. Generate code by double-clicking one of the blue buttons above. An HTML file named BuildInfo.html is created.
2. Select **Open BuildInfo.html** to view the file in a Web browser. Follow the hyperlinks to open the source files listed.

Notes

1. Use `get_param('rtwdemo_buildinfo', 'PostCodeGenCommand')` to see the function executed in the Post Code Gen Command stage.
2. Select **Open rtwdemo_buildinfo_data.m** to study the API for the Build Info object.
3. Select **Open rtwmakecfg.m** to study the use of rtwmakecfg API.
4. The buildInfo object is saved out to the following location `rtwdemo_<target>_rtw/buildInfo.mat`
5. The `packNGo` feature of the buildInfo object is invoked at the end of the post code generation function `rtwdemo_buildinfo_data.m`

[Build Info Support Documentation](#)

[rtwmakecfg API Documentation](#)

[Build Info API Documentation](#)

Copyright 1994-2012 The MathWorks, Inc.

Generate Code from Model

Double-click on the **Generate Code Using Simulink Coder** button to generate code for the GRT target.

Or, if Embedded Coder is installed, double-click on the **Generate Code Using Embedded Coder** button to generate code for the ERT target.

The build process generates a `BuildInfo.html` file to document the build information object.

Examine the Build Process Customizations and Output

Use the links in the model to examine the build process customizations and the post code generation query of the the build information object.

To view the `BuildInfo.html` file in a Web browser, click on **Open BuildInfo.html**.

The example uses the `PostCodeGenCommand` parameter of the model to generate the html file from the build information object. The file provides hyperlinks to open the source files (generated code) from the model. To view the `PostCodeGenCommand` parameter value, type:

```
get_param('rtwdemo_buildinfo', 'PostCodeGenCommand');
```

This value indicates a function to execute in the **Post Code Gen Command** stage.

```
rtwdemo_buildinfo_data(buildInfo);
```

To study how the example uses the `rtwmakecfg` API, click on **Open rtwmakecfg.m** or type:

```
edit rtwmakecfg.m;
```

To study the API for the `buildInfo.mat` object, click on **Open rtwdemo_buildinfo_data.m** or type:

```
edit rtwdemo_buildinfo_data.m;
```

The `buildInfo.mat` object is available at:

```
rtwdemo_<target>_rtw\buildInfo.mat
```

At the end of the `rtwdemo_buildinfo_data.m` post code generation function, the function invokes `packNGO` to package the source and objects identified in the `buildInfo` object for relocation.

Further Study Topics

- “Build Process Customization”
- “Customize Post-Code-Generation Build Processing”
- “Use `rtwmakecfg.m` API to Customize Generated Makefiles”
- “Relocate Code to Another Development Environment”

Suppress Makefile Generation

The code generator provides the ability to suppress makefile generation during the build process. For example, you might do this to integrate tools into the build process that are not driven by makefiles.

To instruct the code generator to not produce a makefile, do one of the following:

- Clear the **Generate makefile** option on the **Code Generation** pane of the Configuration Parameters dialog box.
- Set the value of the configuration parameter `GenerateMakefile` to off.

When you suppress makefile generation,

- You cannot explicitly specify a make command or template makefile.
- You must specify your own instructions for a post code generation processing, including compilation and linking, in a post code generation command as explained in “Program a Post Code Generation Command” on page 35-16 and “Define a Post Code Generation Command” on page 35-17.

Configure Generated Code with TLC

In this section...

“About Configuring Generated Code with TLC” on page 35-22

“Assigning Target Language Compiler Variables” on page 35-22

“Set Target Language Compiler Options” on page 35-23

About Configuring Generated Code with TLC

You can use the Target Language Compiler (TLC) to fine tune your generated code. TLC supports extended code generation variables and options in addition to parameters available on the **Code Generation** pane on the Configuration Parameters dialog box. There are two ways to set TLC variables and options, as described in this section.

Note: You should not customize TLC files in the folder *matlabroot/rtw/c/tlc* even though the capability exists to do so. Such TLC customizations might not be applied during the code generation process and can lead to unpredictable results.

Assigning Target Language Compiler Variables

The `%assign` statement lets you assign a value to a TLC variable, as in

```
%assign MaxStackSize = 4096
```

This is also known as creating a *parameter name/parameter value pair*.

For a description of the `%assign` statement see “Target Language Compiler Directives”. You should write your `%assign` statements in the **Configure RTW code generation settings** section of the system target file.

The following table lists the code generation variables you can set with the `%assign` statement.

Target Language Compiler Optional Variables

Variable	Description
MaxStackSize=N	When the Enable local block outputs check box is selected, the total allocation size of local variables that

Variable	Description
	<p>are declared by block outputs in the model cannot exceed <code>MaxStackSize</code> (in bytes). <code>MaxStackSize</code> can be a positive integer. If the total size of local block output variables exceeds this maximum, the remaining block output variables are allocated in global, rather than local, memory. The default value for <code>MaxStackSize</code> is <code>rtInf</code>, that is, unlimited stack size.</p> <p>Note: Local variables in the generated code from sources other than local block outputs, such as from a Stateflow diagram or MATLAB Function block, and stack usage from sources such as function calls and context switching are not included in the <code>MaxStackSize</code> calculation. For overall executable stack usage metrics, do a target-specific measurement by using run-time (empirical) analysis or static (code path) analysis with object code.</p>
<code>MaxStackVariableSize=N</code>	<p>When the Enable local block outputs check box is selected, this limits the size of a local block output variable declared in the code to N bytes, where $N > 0$. A variable whose size exceeds <code>MaxStackVariableSize</code> is allocated in global, rather than local, memory. The default is 4096.</p>
<code>WarnNonSaturatedBlocks=value</code>	<p>Flag to control display of overflow warnings for blocks that have saturation capability, but have it turned off (unchecked) in their dialog. These are the options:</p> <ul style="list-style-type: none"> • 0 — Warning is not displayed. • 1 — Displays one warning for the model during code generation • 2 — Displays one warning that contains a list of offending blocks

Set Target Language Compiler Options

You can specify TLC command line options for code generation using the model parameter `TLCOptions` in a `set_param` function call. For information about these options, see “Specify TLC Options” on page 25-22 and “Configure TLC”.

Use `rtwmakecfg.m` API to Customize Generated Makefiles

Both the toolchain approach and the template makefile approach for builds let you add the following items to generated makefiles:

- Source folders
- Include folders
- Library names
- Module objects

In this section...

“About the `rtwmakecfg` Function” on page 35-24

“Create the `rtwmakecfg` Function” on page 35-25

“Modify the Template Makefile for `rtwmakecfg`” on page 35-27

About the `rtwmakecfg` Function

Using an `rtwmakecfg` function, you add this information to the makefile during the build operation for S-functions. The `rtwmakecfg` function is particularly useful when specifying added sources and libraries to build a model that contains one or more of your S-function blocks.

To add information pertaining to an S-function to the makefile:

- 1 Create the MATLAB language `rtwmakecfg` function in the `rtwmakecfg.m` file. The code generator associates this file with your S-function based on its folder location. “Create the `rtwmakecfg` Function” on page 35-25 describes the requirements for the `rtwmakecfg` function and the data it returns.
- 2 If you are using the template makefile approach, modify your target's TMF such that it supports macro expansion for the information that the `rtwmakecfg` function returns. “Modify the Template Makefile for `rtwmakecfg`” on page 35-27 describes the required modifications. If you are using the toolchain approach, the information that the `rtwmakecfg` function returns is used by the generated makefile; no further configuration is required.

After the TLC phase of the build process, when generating a makefile, the code generator searches for an `rtwmakecfg.m` file in the folder that contains the S-function MEX file. If it finds the file, the build process calls the `rtwmakecfg` function.

Create the `rtwmakecfg` Function

Create the `rtwmakecfg.m` file containing the `rtwmakecfg` function in the same folder as your S-function component (a MEX-file with a platform-dependent extension, such as `.mexext` on Microsoft Windows systems). The function must return a structured array that contains these fields.

Field	Description
<code>makeInfo.includePath</code>	A cell array that specifies additional include folder names, organized as a row vector. The build process expands the folder names into include instructions in the generated makefile.
<code>makeInfo.sourcePath</code>	A cell array that specifies additional source folder names, organized as a row vector. You must include the folder names of files entered into the S-function modules field on the S-Function Block Parameters dialog box or into the block's <code>SFunctionModules</code> parameter if they are not in the same folder as the S-function. The build process expands the folder names into make rules in the generated makefile.
<code>makeInfo.sources</code>	A cell array that specifies additional source file names (C or C++), organized as a row vector. Do not include the name of the S-function or any files entered into the S-function modules field on the S-Function Block Parameters dialog box or into the block's <code>SFunctionModules</code> parameter. The build process expands the file names into make variables that contain the source files. Specify only file names (with extension). Specify path information with the <code>sourcePath</code> field.
<code>makeInfo.linkLibsObjs</code>	A cell array that specifies additional, fully qualified paths to object or library files against which the generated code links. The build process does not compile the specified objects and libraries. However, it includes them when linking the final executable. This inclusion can be useful for incorporating libraries that you do not want the build process to recompile or for which the source files are not available. You can also use this element to incorporate source files from languages other than C and C++. This is possible if you first create a C compatible object file or library outside of the build process.

Field	Description
<code>makeInfo.precompile</code>	A Boolean flag that indicates whether the libraries specified in the <code>rtwmakecfg.m</code> file exist in a specified location (<code>precompile==1</code>) or if you must create the libraries in the build folder during the build process (<code>precompile==0</code>).
<code>makeInfo.library</code>	A structure array that specifies additional run-time libraries and module objects, organized as a row vector. The build process expands the information into make rules in the generated makefile. For a list of the library fields, see the next table.

The `makeInfo.library` field consists of the following elements.

Element	Description
<code>makeInfo.library(n).Name</code>	A character array that specifies the name of the library (without an extension).
<code>makeInfo.library(n).Location</code>	A character array that specifies the folder in which the library is located when precompiled. For more information, see the description of <code>makeInfo.precompile</code> in the preceding table. A target can use the <code>TargetPreComLibLocation</code> parameter to override this value. See “Specify the Location of Precompiled Libraries” on page 35-9.
<code>makeInfo.library(n).Modules</code>	A cell array that specifies the C or C++ source file base names (without an extension) that comprise the library. Do not include the file extension. The makefile appends the object extension.

Note: The `makeInfo.library` field must fully specify each library and how to build it. The modules list in the `makeInfo.library(n).Modules` element cannot be empty. If you need to specify a link-only library, use the `makeInfo.linkLibsObjs` field instead.

Example:

```
disp(['Running rtwmakecfg from folder: ',pwd]);
makeInfo.includePath = { fullfile(pwd, 'somedir2') };
makeInfo.sourcePath = {fullfile(pwd, 'somedir2'), fullfile(pwd, 'somedir3')};
makeInfo.sources = { 'src1.c', 'src2.cpp'};
```

```

makeInfo.linkLibsObjs = { fullfile(pwd, 'somedir3', 'src3.object'),...
                        fullfile(pwd, 'somedir4', 'mylib.library')};
makeInfo.precompile = 1;
makeInfo.library(1).Name      = 'myprecompiledlib';
makeInfo.library(1).Location = fullfile(pwd, 'somedir2', 'lib');
makeInfo.library(1).Modules  = {'srcfile1' 'srcfile2' 'srcfile3' };

```

Note: If a path that you specify in the `rtwmakecfg.m` API contains spaces, the build process does not convert the path to its nonspace equivalent. If the build environments you intend to support do not support spaces in paths, refer to “Enable Build Process When Folder Names Have Spaces” on page 19-65.

Modify the Template Makefile for `rtwmakecfg`

To expand the information that an `rtwmakecfg` function generates, modify the following sections of your target's TMF:

- Include Path
- C Flags and/or Additional Libraries
- Rules

It is possible that these TMF code examples do not apply to your make utility. For additional examples, see the GRT or ERT TMFs located in *matlabroot/rtw/c/grt* (open) or *matlabroot/rtw/c/ert* (open).

Add Folder Names to the Makefile Include Path

The following TMF code example adds folder names to the include path in the generated makefile:

```

ADD_INCLUDES = \
|>START_EXPAND_INCLUDES<|   -I|>EXPAND_DIR_NAME<| \
|>END_EXPAND_INCLUDES<|

```

Additionally, the `ADD_INCLUDES` macro must be added to the `INCLUDES` line.

```
INCLUDES = -I. -I.. $(MATLAB_INCLUDES) $(ADD_INCLUDES) $(USER_INCLUDES)
```

Add Library Names to the Makefile

The following TMF code example adds library names to the generated makefile.

```
LIBS =
```

```
|>START_PRECOMP_LIBRARIES<|
LIBS += |>EXPAND_LIBRARY_NAME<|.a |>END_PRECOMP_LIBRARIES<|
|>START_EXPAND_LIBRARIES<|
LIBS += |>EXPAND_LIBRARY_NAME<|.a |>END_EXPAND_LIBRARIES<|
```

For more information, see “Control Library Location and Naming During Build” on page 35-7.

Add Rules to the Makefile

The TMF code example adds rules to the generated makefile.

```
|>START_EXPAND_RULES<|
$(BLD)/%.o: |>EXPAND_DIR_NAME<|/%.c $(SRC)/$(MAKEFILE) rtw_proj.tmw
    @$(BLANK)
    @echo ### "|>EXPAND_DIR_NAME<|\\"$*.c"
    $(CC) $(CFLAGS) $(APP_CFLAGS) -o $(BLD)$(DIRCHAR)$*.o \
    |>EXPAND_DIR_NAME<|$(DIRCHAR)$*.c > $(BLD)$(DIRCHAR)$*.lst
|>END_EXPAND_RULES<|

|>START_EXPAND_LIBRARIES<|MODULES_|>EXPAND_LIBRARY_NAME<| = \
|>START_EXPAND_MODULES<| |>EXPAND_MODULE_NAME<|.o \
|>END_EXPAND_MODULES<|

|>EXPAND_LIBRARY_NAME<|.a : $(MAKEFILE) rtw_proj.tmw
$(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
    @$(BLANK)
    @echo ### Creating $@
    $(AR) -r $@ $(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
|>END_EXPAND_LIBRARIES<|

|>START_PRECOMP_LIBRARIES<|MODULES_|>EXPAND_LIBRARY_NAME<| = \
|>START_EXPAND_MODULES<| |>EXPAND_MODULE_NAME<|.o \
|>END_EXPAND_MODULES<|

|>EXPAND_LIBRARY_NAME<|.a : $(MAKEFILE) rtw_proj.tmw
$(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
    @$(BLANK)
    @echo ### Creating $@
    $(AR) -r $@ $(MODULES_|>EXPAND_LIBRARY_NAME<|:%.o=$(BLD)/%.o)
|>END_PRECOMP_LIBRARIES<|
```

Customize Build Process with `STF_make_rtw_hook` File

The build process lets you supply optional custom code in hook methods that are executed at specified points in the code generation and make process. You can use hook methods to add target-specific actions to the build process.

In this section...

“The `STF_make_rtw_hook` File” on page 35-29

“Conventions for Using the `STF_make_rtw_hook` File” on page 35-29

“`STF_make_rtw_hook.m` Function Prototype and Arguments” on page 35-30

“Applications for `STF_make_rtw_hook.m`” on page 35-33

“Control Code Regeneration Using `STF_make_rtw_hook.m`” on page 35-34

“Use `STF_make_rtw_hook.m` for Your Build Procedure” on page 35-35

The `STF_make_rtw_hook` File

You can modify hook methods in a file generically referred to as `STF_make_rtw_hook.m`, where *STF* is the name of a system target file, such as `ert` or `mytarget`. This file implements a function, `STF_make_rtw_hook`, that dispatches to a specific action, depending on the `hookMethod` argument passed in.

The build process calls `STF_make_rtw_hook`, passing in the `hookMethod` argument and other arguments. You implement only those hook methods that your build process requires.

If your model contains reference models, you can implement an `STF_make_rtw_hook.m` for each reference model as required. The build process calls each `STF_make_rtw_hook` for reference models, processing these files recursively (in dependency order).

Conventions for Using the `STF_make_rtw_hook` File

For the build process to call the `STF_make_rtw_hook`, check that the following conditions are met:

- The `STF_make_rtw_hook.m` file is on the MATLAB path.

- The file name is the name of your system target file (STF), appended to the text `_make_rtw_hook.m`. For example, if you generate code with a custom system target file `mytarget.tlc`, name your hook file `mytarget_make_rtw_hook.m`, and name the hook function implemented within the file `mytarget_make_rtw_hook`.
- The hook function implemented in the file uses the function prototype described in “STF_make_rtw_hook.m Function Prototype and Arguments” on page 35-30.

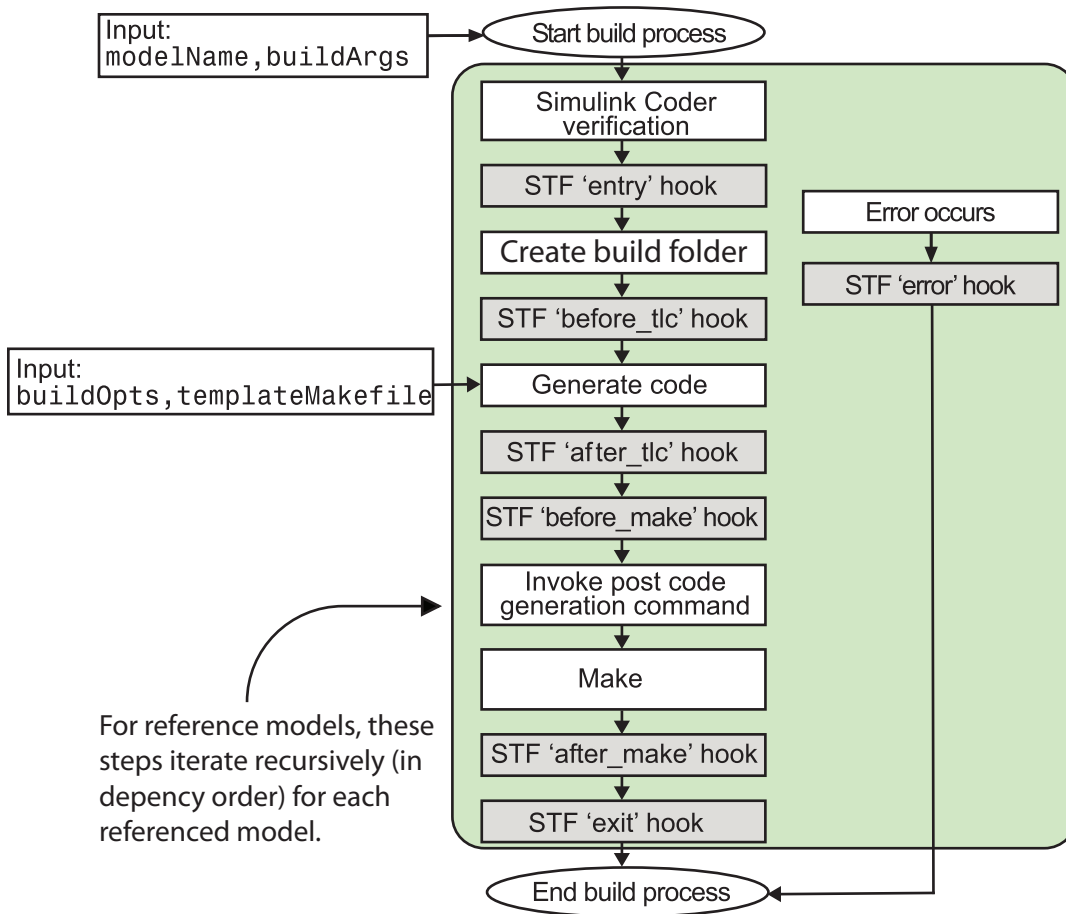
STF_make_rtw_hook.m Function Prototype and Arguments

The function prototype for `STF_make_rtw_hook` is:

```
function STF_make_rtw_hook(hookMethod, modelName, rtwRoot, templateMakefile,  
buildOpts, buildArgs, buildInfo)
```

The arguments are defined as:

- `hookMethod`: Character vector specifying the stage of build process from which the `STF_make_rtw_hook` function is called. The following flowchart summarizes the build process, highlighting the hook points. Valid values for `hookMethod` are `'entry'`, `'before_tlc'`, `'after_tlc'`, `'before_make'`, `'after_make'`, `'exit'`, and `'error'`. The `STF_make_rtw_hook` function dispatches to the relevant code with a `switch` statement.



- **modelName**: Character vector specifying the name of the model. Valid at all stages of the build process.
- **rtwRoot**: Reserved.
- **templateMakefile**: Name of template makefile.
- **buildOpts**: A MATLAB structure containing the fields described in the following list. Valid for the 'before_make', 'after_make', and 'exit' stages only. The buildOpts fields include:
 - **modules**: Character vector specifying a list of additional files to compile.

- **codeFormat**: Character vector specifying the value of the **CodeFormat** TLC variable for the target. (ERT-based targets must use the 'Embedded-C' value for the **CodeFormat** TLC variable and use the corresponding 'ert.tlc' value in the `rtwgensettings.DerivedFrom` field.)
- **noninlinedSFcns**: Cell array specifying a list of noninlined S-functions in the model.
- **compilerEnvVal**: Character vector specifying the compiler environment variable value (for example, `C:\Applications\Microsoft Visual`).
- **buildArgs**: Character vector containing the argument to `make_rtw`. When you invoke the build process, **buildArgs** is copied from the argument following "make_rtw" in the **Configuration Parameters+Code Generation+Make command** field.

For example, the following make arguments from the **Make command** field

```
make_rtw VAR1=0 VAR2=4
```

generate the following:

```
% make -f untitled.mk VAR1=0 VAR2=4
```

The **buildArgs** argument does not apply for toolchain approach builds because these builds do not allow adding make arguments to the `make_rtw` call. To provide custom definitions (for example, `VAR1=0 VAR2=4`) on the compiler command line that apply for both TMF approach and toolchain approach builds, use the **Configuration Parameters > Code Generation > Custom Code > Defines** field.

- **buildInfo**: The MATLAB structure that contains the model build information fields. Valid for the 'after_tlc', 'before_make', 'after_make', and 'exit' stages only. For information about these fields and functions to access them, see "Build Process Customization".

Applications for STF_make_rtw_hook.m

Here are some examples of how you might apply the *STF_make_rtw_hook.m* hook methods.

In general, you can use the 'entry' hook to initialize the build process, for example, to change or validate settings before code is generated. One application for the 'entry' hook is to rerun the auto-configuration script that initially ran at target selection time to compare model parameters before and after the script executes, for validation purposes.

The other hook points, 'before_tlc', 'after_tlc', 'before_make', 'after_make', 'exit', and 'error' are useful for interfacing with external tool chains, source control tools, and other environment tools.

For example, you could use the *STF_make_rtw_hook.m* file at a stage after 'entry' to obtain the path to the build folder. At the 'exit' stage, you could then locate generated code files within the build folder and check them into your version control system. You might use 'error' to clean up static or global data used by the hook function when an error occurs during code generation or the build process.

Note: The build process temporarily changes the MATLAB working folder to the build folder for stages 'before_make', 'after_make', 'exit', and 'error'. Your *STF_make_rtw_hook.m* file must not make incorrect assumptions about the location of the build folder. At a point after the 'entry' stage, you can obtain the path to the build folder. In the following MATLAB code example, the build folder path is returned as a character vector to the variable `buildDirPath`.

```
buildDirPath = rtwprivate('get_makertwsettings',gcs,'BuildDirectory');
```

Control Code Regeneration Using `STF_make_rtw_hook.m`

When you rebuild a model, by default, the build process performs checks to determine whether changes to the model or relevant settings require regeneration of the top model code. (For details on the criteria, see “Control Regeneration of Top Model Code” on page 25-43.) If the checks determine that top model code generation is required, the build process fully regenerates and compiles the model code. If the checks indicate that the top model generated code is current with respect to the model, and model settings do not require full regeneration, the build process omits regeneration of the top model code.

Regardless of whether the top model code is regenerated, the build process subsequently calls the build process hooks, including `STF_make_rtw_hook` functions and the post code generation command. The following mechanisms allow you to perform actions related to code regeneration in the `STF_make_rtw_hook` functions:

- To force code regeneration, use the following function call from the 'entry' hook:

```
rtw.targetNeedsCodeGen('set', true);
```
- In hooks from 'before_tlc' through 'exit', the `buildOpts` structure passed to the hook has a Boolean field `codeWasUpToDate`. The field is set to `true` if model code was up to date and code was not regenerated, or `false` if code was not up to date and code was regenerated. You can customize hook actions based on the value of this field. For example:

```
...
case 'before_tlc'
    if buildOpts.codeWasUpToDate
        %Perform hook actions for up to date model
    else
        %Perform hook actions for full code generation
    end
...

```

Use STF_make_rtw_hook.m for Your Build Procedure

To create a custom *STF_make_rtw_hook* hook file for your build procedure, copy and edit the example `ert_make_rtw_hook.m` file, which is located in the folder `matlabroot/toolbox/coder/embeddedcoder` (open), as follows:

- 1** Copy `ert_make_rtw_hook.m` to a folder in the MATLAB path. Rename it in accordance with the naming conventions described in “Conventions for Using the STF_make_rtw_hook File” on page 35-29. For example, to use it with the GRT target `grt.tlc`, rename it to `grt_make_rtw_hook.m`.
- 2** Rename the `ert_make_rtw_hook` function within the file to match the file name.
- 3** Implement the hooks that you require by adding code to case statements within the `switch hookMethod` statement.

Customize Build Process with `sl_customization.m`

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

In this section...

“The `sl_customization.m` File” on page 35-36

“Register Build Process Hook Functions Using `sl_customization.m`” on page 35-38

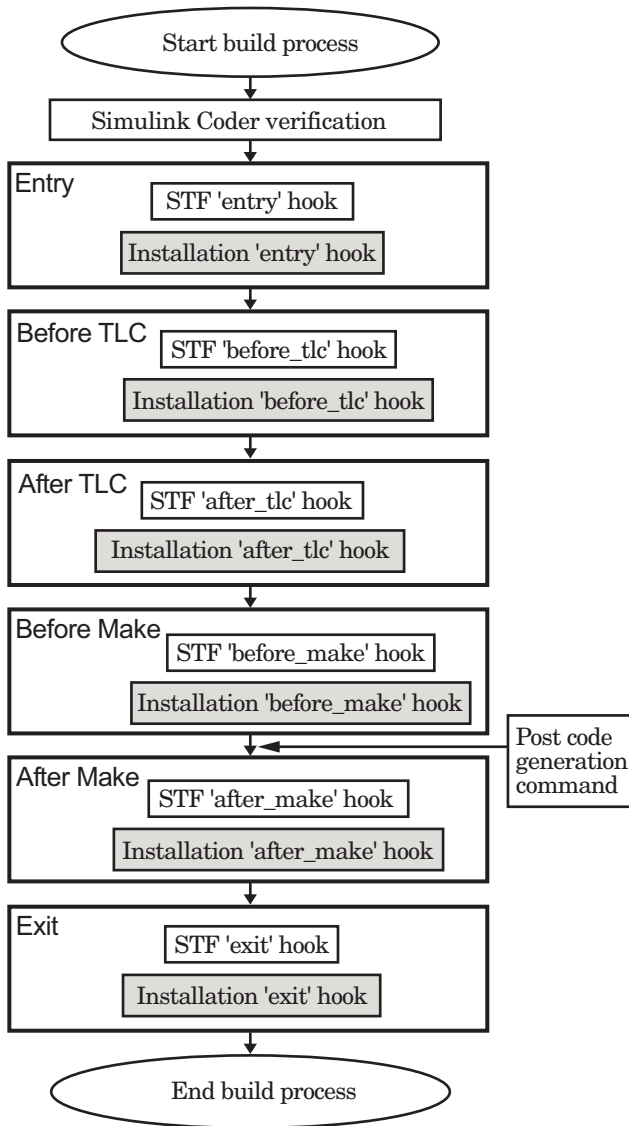
“Variables Available for `sl_customization.m` Hook Functions” on page 35-38

“Example Build Process Customization Using `sl_customization.m`” on page 35-39

The `sl_customization.m` File

The `sl_customization.m` file can be used to register installation-specific hook functions to be invoked during the build process. The hook functions that you register through `sl_customization.m` complement System Target File (STF) hooks (described in “Customize Build Process with `STF_make_rtw_hook` File” on page 35-29) and post-code generation commands (described in “Customize Post-Code-Generation Build Processing” on page 35-14).

The following figure shows the relationship between installation-level hooks and the other available mechanisms for customizing the build process.



Register Build Process Hook Functions Using `sl_customization.m`

To register installation-level hook functions that will be invoked during the build process, you create a MATLAB function called `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function accepts one argument: a handle to a customization manager object. For example,

```
function sl_customization(cm)
```

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

```
hObj = cm.RTWBuildCustomizer;
```

You then invoke methods to register your customizations. The customization manager object includes the following method for registering build process hook customizations:

- `addUserHook(hObj, hookType, hook)`

Registers the MATLAB hook script or function specified by `hook` for the build process stage represented by `hookType`. The valid values for `hookType` are `'entry'`, `'before_tlc'`, `'after_tlc'`, `'before_make'`, `'after_make'`, and `'exit'`.

Your instance of the `sl_customization` function should use this method to register installation-specific hook functions.

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, you must restart the Simulink session or enter the following command in the Command Window to enable the changes:

```
sl_refresh_customizations
```

Variables Available for `sl_customization.m` Hook Functions

The following variables are available for `sl_customization.m` hook functions to use:

- `modelName` — The name of the Simulink model (valid for all stages)
- `dependencyObject` — An object containing the dependencies of the generated code (valid only for the `'after_make'` stage)

A hook script can directly access the valid variables. A hook function can pass the valid variables as arguments to the function. For example:


```
hObj.addUserHook('after_make', 'afterMakeFunction(modelName,dependencyObject);');
```

Example Build Process Customization Using sl_customization.m

The `sl_customization.m` file shown in Example 1: `sl_customization.m` for Build Process Customizations uses the `addUserHook` method to specify installation-specific build process hooks to be invoked at the 'entry' and 'after_tlc' stages of the build process. For the hook function source code, see Example 2: `CustomRTWEntryHook.m` and Example 3: `CustomRTWPostProcessHook.m`.

Example 1: sl_customization.m for Build Process Customizations

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

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

% Register build process hooks
hObj.addUserHook('entry', 'CustomRTWEntryHook(modelName);');
hObj.addUserHook('after_tlc', 'CustomRTWPostProcessHook(modelName);');

end
```

Example 2: CustomRTWEntryHook.m

```
function [str, status] = CustomRTWEntryHook(modelName)
str =sprintf('Custom entry hook for model '%s.',modelName);
disp(str)
status =1;
```

Example 3: CustomRTWPostProcessHook.m

```
function [str, status] = CustomRTWPostProcessHook(modelName)
str =sprintf('Custom post process hook for model '%s.',modelName);
disp(str)
status =1;
```

If you include the above three files on the MATLAB path of the Simulink installation that you want to customize, the coded hook function messages will appear in the displayed output for builds. For example, if you open the ERT-based model `rtwdemo_udt`, open the **Code Generation** pane of the Configuration Parameters dialog box, and press **Ctrl+B** to initiate a build, the following messages are displayed:

```
>> rtwdemo_udt

### Starting build procedure for model: rtwdemo_udt
Custom entry hook for model 'rtwdemo_udt.'
Custom post process hook for model 'rtwdemo_udt.'
```

```
### Successful completion of build procedure for model: rtwdemo_udt  
>>
```

Replace STF_rtw_info_hook Supplied Target Data

Prior to MATLAB Release 14, custom targets supplied target-specific information with a hook file (referred to as *STF_rtw_info_hook.m*). The *STF_rtw_info_hook* specified properties such as word sizes for integer data types (for example, `char`, `short`, `int`, and `long`), and C implementation-specific properties of the custom target.

The *STF_rtw_info_hook* mechanism has been replaced by the **Hardware Implementation** pane of the Configuration Parameters dialog box. Using this dialog box, you can specify properties that were formerly specified in your *STF_rtw_info_hook* file.

For backward compatibility, existing *STF_rtw_info_hook* files are available. However, you should convert your target and models to use of the **Hardware Implementation** pane. See “Configure Target Hardware” on page 10-3.

Customize Build to Use Shared Utility Code

The shared utility folders (`slprj/target/_sharedutils`) typically store generated utility code that is common to a top model and the models it references. You can also force the build process to use a shared utilities folder for a standalone model. See “Specify Target Interfaces” on page 17-10 for details.

If you want your target to support compilation of code generated in the shared utilities folder, you must modify your template makefile (TMF). The shared utilities folder is required to support model reference builds. See “Support Model Referencing” on page 37-83 to learn about additional updates for supporting model reference builds.

The exact syntax of the changes can vary due to differences in the `make` utility and compiler/archive tools used by your target. The examples below are based on the Free Software Foundation's GNU `make` utility. You can find the following updated TMF examples for GNU and Microsoft Visual C++ `make` utilities in the GRT and ERT target folders:

- GRT: `matlabroot/rtw/c/grt` (open)
 - `grt_lcc.tmf`
 - `grt_vc.tmf`
 - `grt_unix.tmf`
- ERT: `matlabroot/rtw/c/ert` (open)
 - `ert_lcc.tmf`
 - `ert_vc.tmf`
 - `ert_unix.tmf`

Use the GRT or ERT examples as a guide to the location, within the TMF, of the changes and additions described below.

Note The ERT-based TMFs contain extra code to handle generation of ERT S-functions and model reference simulation targets. Your target does not need to handle these cases.

Modify Template Makefiles to Support Shared Utilities

Make the following changes to your TMF to support the shared utilities folder:

- 1 Add the following make variables and tokens to be expanded when the makefile is generated:

```
SHARED_SRC      = |>SHARED_SRC<|
SHARED_SRC_DIR  = |>SHARED_SRC_DIR<|
SHARED_BIN_DIR  = |>SHARED_BIN_DIR<|
SHARED_LIB      = |>SHARED_LIB<|
```

SHARED_SRC specifies the shared utilities folder location and the source files in it. A typical expansion in a makefile is

```
SHARED_SRC      = ../slprj/ert/_sharedutils/*.c
```

SHARED_LIB specifies the library file built from the shared source files, as in the following expansion.

```
SHARED_LIB      = ../slprj/ert/_sharedutils/rtwshared.lib
```

SHARED_SRC_DIR and **SHARED_BIN_DIR** allow specification of separate folders for shared source files and the library compiled from the source files. In the current release, TMFs use the same path, as in the following expansions.

```
SHARED_SRC_DIR  = ../slprj/ert/_sharedutils
SHARED_BIN_DIR  = ../slprj/ert/_sharedutils
```

- 2 Set the **SHARED_INCLUDES** variable according to whether shared utilities are in use. Then append it to the overall **INCLUDES** variable.

```
SHARED_INCLUDES =
ifneq ($(SHARED_SRC_DIR),)
SHARED_INCLUDES = -I$(SHARED_SRC_DIR)
endif
```

```
INCLUDES = -I. $(MATLAB_INCLUDES) $(ADD_INCLUDES) \
$(USER_INCLUDES) $(SHARED_INCLUDES)
```

- 3 Update the **SHARED_SRC** variable to list shared files explicitly.

```
SHARED_SRC := $(wildcard $(SHARED_SRC))
```

- 4 Create a **SHARED_OBJS** variable based on **SHARED_SRC**.

```
SHARED_OBJS = $(addsuffix .o, $(basename $(SHARED_SRC)))
```

- 5 Create an **OPTS** (options) variable for compilation of shared utilities.

```
SHARED_OUTPUT_OPTS = -o $@
```

- 6 Provide a rule to compile the shared utility source files.

```
$(SHARED_OBJS) : $(SHARED_BIN_DIR)/%.o : $(SHARED_SRC_DIR)/%.c
$(CC) -c $(CFLAGS) $(SHARED_OUTPUT_OPTS) $<
```

- 7** Provide a rule to create a library of the shared utilities. The following example is based on The Open Group UNIX platforms.

```
$(SHARED_LIB) : $(SHARED_OBJS)
    @echo "### Creating $@"
    ar r $@ $(SHARED_OBJS)
    @echo "### Created $@"
```

- 8** Add `SHARED_LIB` to the rule that creates the final executable.

```
$(PROGRAM) : $(OBJS) $(LIBS) $(SHARED_LIB)
$(LD) $(LDFLAGS) -o $@ $(LINK_OBJS) $(LIBS) $(SHARED_LIB) \
    $(SYSLIBS)
@echo "### Created executable: $(MODEL)"
```

- 9** Remove explicit reference to `rt_nonfinite.c` or `rt_nonfinite.cpp` from your TMF. For example, change

```
ADD_SRCS = $(RTWLOG) rt_nonfinite.c
```

to

```
ADD_SRCS = $(RTWLOG)
```

Run-Time Data Interface Extensions in Simulink Coder

- “Customize Generated ASAP2 File” on page 36-2
- “Create a Transport Layer for External Communication” on page 36-8

Customize Generated ASAP2 File

In this section...

- “About ASAP2 File Customization” on page 36-2
- “ASAP2 File Structure on the MATLAB Path” on page 36-2
- “Customize the Contents of the ASAP2 File” on page 36-3
- “ASAP2 Templates” on page 36-4
- “Customize Computation Method Names” on page 36-6
- “Suppress Computation Methods for FIX_AXIS” on page 36-7

About ASAP2 File Customization

The Embedded Coder product provides a number of Target Language Compiler (TLC) files to enable you to customize the ASAP2 file generated from a Simulink model.

ASAP2 File Structure on the MATLAB Path

The ASAP2 related files are organized within the folders identified below:

- TLC files for generating ASAP2 file

The *matlabroot/rtw/c/tlc/mw* (open) folder contains TLC files that generate ASAP2 files, *asamlib.tlc*, *asap2lib.tlc*, *asap2main.tlc*, and *asap2group1ib.tlc*. These files are included by the selected **System target file**. (See “Targets Supporting ASAP2” on page 29-3.)

- ASAP2 target files

The *matlabroot/toolbox/rtw/targets/asap2/asap2* (open) folder contains the ASAP2 system target file and other control files.

- Customizable TLC files

The *matlabroot/toolbox/rtw/targets/asap2/asap2/user* (open) folder contains files that you can modify to customize the content of your ASAP2 files.

- ASAP2 templates

The `matlabroot/toolbox/rtw/targets/asap2/asap2/user/templates` (open) folder contains templates that define each type of CHARACTERISTIC in the ASAP2 file.

Customize the Contents of the ASAP2 File

The ASAP2 related TLC files enable you to customize the appearance of the ASAP2 file generated from a Simulink model. Most customization is done by modifying or adding to the files contained in the `matlabroot/toolbox/rtw/targets/asap2/asap2/user` (open) folder. This section refers to this folder as the `asap2/user` folder.

The user-customizable files provided are divided into two groups:

- The *static* files define the parts of the ASAP2 file that are related to the environment in which the generated code is used. They describe information specific to the user or project. The static files are not model dependent.
- The *dynamic* files define the parts of the ASAP2 file that are generated based on the structure of the source model.

The procedure for customizing the ASAP2 file is as follows:

- 1 Make a copy of the `asap2/user` folder before making modifications.
- 2 Remove the old `asap2/user` folder from the MATLAB path, or add the new `asap2/user` folder to the MATLAB path above the old folder. The MATLAB session uses the ASAP2 setup file, `asap2setup.tlc`, in the new folder.

`asap2setup.tlc` specifies the folders and files to include in the TLC path during the ASAP2 file generation process. Modify `asap2setup.tlc` to control the folders and folders included in the TLC path.

- 3 Modify the static parts of the ASAP2 file. These include
 - Project and header symbols, which are specified in `asap2setup.tlc`
 - Static sections of the file, such as file header and tail, A2ML, MOD_COMMON, and so on. These are specified in `asap2userlib.tlc`.
 - Specify the appearance of the dynamic contents of the ASAP2 file by modifying the existing ASAP2 templates or by defining new ASAP2 templates. Sections of the ASAP2 file affected include

RECORD_LAYOUT: modify parts of the ASAP2 template files.

CHARACTERISTIC: modify parts of the ASAP2 template files. For more information on modifying the appearance of **CHARACTERISTIC** records, see “ASAP2 Templates” on page 36-4.

- **MEASUREMENT:** These are specified in `asap2userlib.tlc`.
- **COMPU_METHOD:** These are specified in `asap2userlib.tlc`.

ASAP2 Templates

The appearance of **CHARACTERISTIC** records in the ASAP2 file is controlled using a different template for each type of **CHARACTERISTIC**. The `asap2/user` folder contains template definition files for scalars, 1-D Lookup Table blocks and 2-D Lookup Table blocks. You can modify these template definition files, or you can create additional templates as required.

The procedure for creating a new ASAP2 template is as follows:

- 1 Create a template definition file. See “Create Template Definition Files” on page 36-4.
- 2 Include the template definition file in the TLC path. The path is specified in the ASAP2 setup file, `asap2setup.tlc`.

Create Template Definition Files

This section describes the components that make up an ASAP2 template definition file. This description is in the form of code examples from `asap2lookup1d.tlc`, the template definition file for the Lookup1D template. This template corresponds to the Lookup1D parameter group.

Note When creating a new template, use the corresponding parameter group name in place of `Lookup1D` in the code shown.

Template Registration Function

The input argument is the name of the parameter group associated with this template:

```
%<LibASAP2RegisterTemplate("Lookup1D")>
```

RECORD_LAYOUT Name Definition Function

Record layout names (aliases) can be arbitrarily specified for each data type. This function is used by the other components of this file.

```
%function ASAP2UserFcnRecordLayoutAlias_Lookup1D(dtId) void
  %switch dtId
    %case tSS_UINT8
      %return "Lookup1D_UBYTE"
    ...
  %endswitch
%endfunction
```

Function to Write RECORD_LAYOUT Definitions

This function writes RECORD_LAYOUT definitions associated with this template. The function is called by the built-in functions involved in the ASAP2 file generation process. The function name must be defined as shown, with the template name after the underscore:

```
%function ASAP2UserFcnWriteRecordLayout_Lookup1D() Output
  /begin RECORD_LAYOUT
  %<ASAP2UserFcnRecordLayoutAlias_Lookup1D(tSS_UINT8)>
  ...
  /end RECORD_LAYOUT
%endfunction
```

Function to Write the CHARACTERISTIC

This function writes the CHARACTERISTIC associated with this template. The function is called by the built-in functions involved in the ASAP2 file generation process. The function name must be defined as shown, with the template name after the underscore.

The input argument to this function is a pointer to a parameter group record. The example shown is for a LOOKUP1D parameter group that has two members. The references to the associated x and y data records are obtained from the parameter group record as shown.

This function calls a number of built-in functions to obtain the required information. For example, LibASAP2GetSymbol returns the symbol (name) for the specified data record:

```
%function ASAP2UserFcnWriteCharacteristic_Lookup1D(paramGroup)
Output
```

```

%assign xParam = paramGroup.Member[0].Reference
%assign yParam = paramGroup.Member[1].Reference
%assign dtId = LibASAP2GetDataTypeId(xParam)
  /begin CHARACTERISTIC
    /* Name */           %<LibASAP2GetSymbol(xParam)>
    /* Long identifier */ %<LibASAP2GetLongID(xParam)>
    ...
  /end CHARACTERISTIC
%endfunction

```

Customize Computation Method Names

In generated ASAP2 files, computation methods translate the electronic control unit (ECU) internal representation of measurement and calibration quantities into a physical model oriented representation. Simulink Coder software provides the ability to customize the names of computation methods. You can provide names that are more intuitive, enhancing ASAP2 file readability, or names that meet organizational requirements.

To customize computation method names, use the MATLAB function `getCompuMethodName`, which is defined in `matlabroot/toolbox/rtw/targets/asap2/asap2/user/getCompuMethodName.m`.

The `getCompuMethodName` function constructs a computation method name. The function prototype is

```
cmName = getCompuMethodName(dataTypeName, cmUnits)
```

where *dataTypeName* is the name of the data type associated with the computation method, *cmUnits* is the units as specified in the `Unit` property of a `Simulink.Parameter` or `Simulink.Signal` object (for example, rpm or m/s), and *cmName* returns the constructed computation method name.

The default constructed name returned by the function has the format

```
<localPrefix><datatype>_<cmUnits>
```

where

- `<local_Prefix>` is a local prefix, `CM_`, defined in `matlabroot/toolbox/rtw/targets/asap2/asap2/user/getCompuMethodName.m`.
- `<datatype>` and `<cmUnits>` are the arguments you specified to the `getCompuMethodName` function.

Additionally, in the generated ASAP2 file, the constructed name is prefixed with `<ASAP2CompuMethodName_Prefix>`, a model prefix defined in `matlabroot/toolbox/rtw/targets/asap2/asap2/user/asap2setup.tlc`.

For example, if you call the `getCompuMethodName` function with the `dataTypeName` argument `'int16'` and the `cmUnits` argument `'m/s'`, and generate an ASAP2 file for a model named `myModel`, the computation method name would appear in the generated file as follows:

```
/begin COMPU_METHOD
  /* Name of CompuMethod */ myModel_CM_int16_m_s
  /* Units */ "m/s"
  ...
/end COMPU_METHOD
```

Suppress Computation Methods for FIX_AXIS

Versions 1.51 and later of the ASAP2 specification state that for certain cases of lookup table axis descriptions (integer data type and no doc units), a computation method is not required and the Conversion Method parameter must be set to the value `NO_COMPU_METHOD`. You can control whether or not computation methods are suppressed when not required using the Target Language Compiler (TLC) option `ASAP2GenNoCompuMethod`. This TLC option is disabled by default. If you enable the option, ASAP2 file generation does not generate computation methods for lookup table axis descriptions when not required, and instead generates the value `NO_COMPU_METHOD`. For example:

```
/begin CHARACTERISTIC
/* Name */
lu1d_fix_axisTable_data
...
/begin AXIS_DESCR
  ...
  /* Conversion Method */
NO_COMPU_METHOD
  ...
/end CHARACTERISTIC
```

The `ASAP2GenNoCompuMethod` option is defined in `matlabroot/toolbox/rtw/targets/asap2/asap2/user/asap2setup.tlc`.

Create a Transport Layer for External Communication

In this section...

“About Creating a Transport Layer for External Communication” on page 36-8

“Design of External Mode” on page 36-8

“External Mode Communications Overview” on page 36-11

“External Mode Source Files” on page 36-12

“Implement a Custom Transport Layer” on page 36-16

About Creating a Transport Layer for External Communication

This section helps you to connect your custom target by using external mode using your own low-level communications layer. The topics include:

- An overview of the design and operation of external mode
- A description of external mode source files
- Guidelines for modifying the external mode source files and building an executable to handle the tasks of the default `ext_comm` MEX-file

This section assumes that you are familiar with the execution of Simulink Coder programs, and with the basic operation of external mode.

Design of External Mode

External mode communication between the Simulink engine and a target system is based on a client/server architecture. The client (the Simulink engine) transmits messages requesting the server (target) to accept parameter changes or to upload signal data. The server responds by executing the request.

A low-level *transport layer* handles physical transmission of messages. Both the Simulink engine and the model code are independent of this layer. Both the transport layer and code directly interfacing to the transport layer are isolated in separate modules that format, transmit, and receive messages and data packets.

This design makes it possible for different targets to use different transport layers. The GRT, ERT, and RSim targets support host/target communication by using TCP/IP and

RS-232 (serial) communication. The Simulink Desktop Real-Time target supports shared memory communication. The Wind River Systems Tornado® target supports TCP/IP only.

The Simulink Coder product provides full source code for both the client and server-side external mode modules, as used by the GRT, ERT, Rapid Simulation, and Tornado targets, and the Simulink Desktop Real-Time and Simulink Real-Time products. The main client-side module is `ext_comm.c`. The main server-side module is `ext_svr.c`.

These two modules call the specified transport layer through the following source files.

Built-In Transport Layer Implementations

Protocol	Client or Server?	Source Files
TCP/IP	Client (host)	<ul style="list-style-type: none"> <code>matlabroot/toolbox/coder/simulinkcoder_core/ext_mode/host/common/rtiostream_interface.c</code> <code>matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c</code>
	Server (target)	<ul style="list-style-type: none"> <code>matlabroot/rtw/c/src/ext_mode/common/rtiostream_interface.c</code> <code>matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c</code>
Serial	Client (host)	<ul style="list-style-type: none"> <code>matlabroot/toolbox/coder/simulinkcoder_core/ext_mode/host/serial/ext_serial_transport.c</code> <code>matlabroot/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c</code>
	Server (target)	<ul style="list-style-type: none"> <code>matlabroot/rtw/c/src/ext_mode/serial/ext_svr_serial_transport.c</code> <code>matlabroot/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c</code>

For serial communication, the modules `ext_serial_transport.c` and `rtiostream_serial.c` implement the client-side transport functions and the modules `ext_svr_serial_transport.c` and `rtiostream_serial.c` implement the corresponding server-side functions. For TCP/IP communication, the modules `rtiostream_interface.c` and `rtiostream_tcpip.c` implement both client-side and server-side functions. You can edit copies of these files (but do not modify the originals).

You can support external mode using your own low-level communications layer by creating similar files using the following templates:

- Client (host) side: `matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c` (TCP/IP) or `matlabroot/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c` (serial)
- Server (target) side: `matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c` (TCP/IP) or `matlabroot/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c` (serial)

The file `rtiostream_interface.c` is an interface between the external mode protocol and an `rtiostream` communications channel. For more details on implementing an `rtiostream` communications channel, see “Communications `rtiostream` API” in the Embedded Coder documentation. Implement your `rtiostream` communications channel by using the documented interface to avoid having to make changes to the file `rtiostream_interface.c` or other external mode related files.

Note Do not modify working source files. Use the templates provided in the `/custom` or `/rtiostream` folder as starting points, guided by the comments within them.

You need only provide code that implements low-level communications. You need not be concerned with issues such as data conversions between host and target, or with the formatting of messages. The Simulink Coder software handles these functions.

On the client (Simulink engine) side, communications are handled by `ext_comm` (for TCP/IP) and `ext_serial_win32_comm` (for serial) MEX-files.

On the server (target) side, external mode modules are linked into the target executable. This takes place automatically if the **External mode** code generation option is selected at code generation time, based on the **External mode transport** option selected in the target code generation options dialog box. These modules, called from the main program and the model execution engine, are independent of the generated model code.

The general procedure for implementing your own client-side low-level transport protocol is as follows:

- 1 Edit the template `rtiostream_tcpip.c` to replace low-level communication calls with your own communication calls.
- 2 Generate a MEX-file executable for your custom transport.

- 3 Register your new transport layer with the Simulink software, so that the transport can be selected for a model using the **Interface** pane of the Configuration Parameters dialog box.

For more details, see “Create a Custom Client (Host) Transport Protocol” on page 36-17.

The general procedure for implementing your own server-side low-level transport protocol is as follows:

- 1 Edit the template `rtiostream_tcpip.c` to replace low-level communication calls with your own communication calls. Typically this involves writing or integrating device drivers for your target hardware.
- 2 Modify template makefiles to support the new transport.

For more details, see “Create a Custom Server (Target) Transport Protocol” on page 36-21.

External Mode Communications Overview

This section gives a high-level overview of how a Simulink Coder generated program communicates with Simulink external mode. This description is based on the TCP/IP version of external mode that ships with the Simulink Coder product.

For communication to take place,

- The server (target) program must have been built with the conditional `EXT_MODE` defined. `EXT_MODE` is defined in the `model.mk` file if the **External mode** code generation option was selected at code generation time.
- Both the server program and the Simulink software must be executing. This does not mean that the model code in the server system must be executing. The server can be waiting for the Simulink engine to issue a command to start model execution.

The client and server communicate by using bidirectional sockets carrying packets. Packets consist either of *messages* (commands, parameter downloads, and responses) or *data* (signal uploads).

If the target program was invoked with the `-w` command-line option, the program enters a wait state until it receives a message from the host. Otherwise, the program begins execution of the model. While the target program is in a wait state, the Simulink engine can download parameters to the target and configure data uploading.

When the user chooses the **Connect to Target** option from the **Simulation** menu, the host initiates a handshake by sending an `EXT_CONNECT` message. The server responds with information about itself. This information includes

- Checksums. The host uses model checksums to determine that the target code is an exact representation of the current Simulink model.
- Data format information. The host uses this information when formatting data to be downloaded, or interpreting data that has been uploaded.

At this point, host and server are connected. The server is either executing the model or in the wait state. (In the latter case, the user can begin model execution by selecting **Start Real-Time Code** from the **Simulation** menu.)

During model execution, the message server runs as a background task. This task receives and processes messages such as parameter downloads.

Data uploading comprises both foreground execution and background servicing of the signal packets. As the target computes model outputs, it also copies signal values into data upload buffers. This occurs as part of the task associated with each task identifier (`tid`). Therefore, data collection occurs in the foreground. Transmission of the collected data, however, occurs as a background task. The background task sends the data in the collection buffers to the Simulink engine by using data packets.

The host initiates most exchanges as messages. The target usually sends a response confirming that it has received and processed the message. Examples of messages and commands are:

- Connection message / connection response
- Start target simulation / start response
- Parameter download / parameter download response
- Arm trigger for data uploading / arm trigger response
- Terminate target simulation / target shutdown response

Model execution terminates when the model reaches its final time, when the host sends a terminate command, or when a Stop Simulation block terminates execution. On termination, the server informs the host that model execution has stopped, and shuts down its socket. The host also shuts down its socket, and exits external mode.

External Mode Source Files

- “Client (Host) MEX-file Interface Source Files” on page 36-13

- “Server (Target) Source Files” on page 36-14
- “Other Files in the Server Folder” on page 36-16

Client (Host) MEX-file Interface Source Files

The source files for the MEX-file interface component are located in the folder *matlabroot/toolbox/coder/simulinkcoder_core/ext_mode/host* (open), except as noted:

- `common/ext_comm.c`

This file is the core of external mode communication. It acts as a relay station between the target and the Simulink engine. `ext_comm.c` communicates to the Simulink engine by using a shared data structure, `ExternalSim`. It communicates to the target by using calls to the transport layer.

Tasks carried out by `ext_comm.c` include establishment of a connection with the target, downloading of parameters, and termination of the connection with the target.

- `common/rtiostream_interface.c`

This file is an interface between the external mode protocol and an `rtiostream` communications channel. For more details on implementing an `rtiostream` communications channel, see “Communications `rtiostream` API” in the Embedded Coder documentation. Implement your `rtiostream` communications channel using the documented interface to avoid having to change the file `rtiostream_interface.c` or other external mode related files.

- `matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c`

This file implements required TCP/IP transport layer functions. The version of `rtiostream_tcpip.c` shipped with the Simulink Coder software uses TCP/IP functions including `recv()`, `send()`, and `socket()`.

- `matlabroot/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c`

This file implements required serial transport layer functions. The version of `rtiostream_serial.c` shipped with the Simulink Coder software uses serial functions including `ReadFile()`, `WriteFile()`, and `CreateFile()`.

- `serial/ext_serial_transport.c`

This file implements required serial transport layer functions.

`ext_serial_transport.c` includes `ext_serial_utils.c`, which is located in `matlabroot/rtw/c/src/ext_mode/serial` (open) and contains functions common to client and server sides.

- `common/ext_main.c`

This file is a MEX-file wrapper for external mode. `ext_main.c` interfaces to the Simulink engine by using the standard `mexFunction` call. (See the `mexFunction` reference page and “MATLAB API for Other Languages” for more information.)

`ext_main.c` contains a function dispatcher, `esGetAction`, that sends requests from the Simulink engine to `ext_comm.c`.

- `common/ext_convert.c` and `ext_convert.h`

This file contains functions used for converting data from host to target formats (and vice versa). Functions include byte-swapping (big to little- endian), conversion from non-IEEE floats to IEEE doubles, and other conversions. These functions are called both by `ext_comm.c` and directly by the Simulink engine (by using function pointers).

Note You do not need to customize `ext_convert` to implement a custom transport layer. However, you might want to customize `ext_convert` for the intended target. For example, if the target represents the `float` data type in Texas Instruments format, `ext_convert` must be modified to perform a Texas Instruments to IEEE conversion.

- `common/extsim.h`

This file defines the `ExternalSim` data structure and access macros. This structure is used for communication between the Simulink engine and `ext_comm.c`.

- `common/extutil.h`

This file contains only conditionals for compilation of the `assert` macro.

- `common/ext_transport.h`

This file defines functions that must be implemented by the transport layer.

Server (Target) Source Files

These files are linked into the `model.exe` executable. They are located within `matlabroot/rtw/c/src/ext_mode` (open) except as noted.

- `common/ext_svr.c`

`ext_svr.c` is analogous to `ext_comm.c` on the host, but generally is responsible for more tasks. It acts as a relay station between the host and the generated code. Like `ext_comm.c`, `ext_svr.c` carries out tasks such as establishing and terminating connection with the host. `ext_svr.c` also contains the background task functions that either write downloaded parameters to the target model, or extract data from the target data buffers and send it back to the host.

- `common/rtiostream_interface.c`

This file is an interface between the external mode protocol and an `rtiostream` communications channel. For more details on implementing an `rtiostream` communications channel, see “Communications `rtiostream` API” in the Embedded Coder documentation. Implement your `rtiostream` communications channel by using the documented interface to avoid having to change the file `rtiostream_interface.c` or other external mode related files.

- `matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c`

This file implements required TCP/IP transport layer functions. The version of `rtiostream_tcpip.c` shipped with the Simulink Coder software uses TCP/IP functions including `recv()`, `send()`, and `socket()`.

- `matlabroot/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c`

This file implements required serial transport layer functions. The version of `rtiostream_serial.c` shipped with the software uses serial functions including `ReadFile()`, `WriteFile()`, and `CreateFile()`.

- `matlabroot/rtw/c/src/rtiostream.h`

This file defines the `rtIOStream*` functions implemented in `rtiostream_tcpip.c`.

- `serial/ext_svr_serial_transport.c`

This file implements required serial transport layer functions. `ext_svr_serial_transport.c` includes `serial/ext_serial_utils.c`, which contains functions common to client and server sides.

- `common/updown.c`

`updown.c` handles the details of interacting with the target model. During parameter downloads, `updown.c` does the work of installing the new parameters into the

model's parameter vector. For data uploading, `updown.c` contains the functions that extract data from the model's `blockio` vector and write the data to the upload buffers. `updown.c` provides services both to `ext_svr.c` and to the model code (for example, `grt_main.c`). It contains code that is called by using the background tasks of `ext_svr.c` as well as code that is called as part of the higher priority model execution.

- `matlabroot/rtw/c/src/dt_info.h` (included by generated model build file `model.h`)

These files contain data type transition information that allows access to multi-data type structures across different computer architectures. This information is used in data conversions between host and target formats.

- `common/updown_util.h`

This file contains only conditionals for compilation of the `assert` macro.

- `common/ext_svr_transport.h`

This file defines the `Ext*` functions that must be implemented by the server (target) transport layer.

Other Files in the Server Folder

- `common/ext_share.h`

Contains message code definitions and other definitions required by both the host and target modules.

- `serial/ext_serial_utils.c`

Contains functions and data structures for communication, MEX link, and generated code required by both the host and target modules of the transport layer for serial protocols.

- The serial transport implementation includes the additional files
 - `serial/ext_serial_pkt.c` and `ext_serial_pkt.h`
 - `serial/ext_serial_port.h`

Implement a Custom Transport Layer

- “Requirements for Custom Transport Layers” on page 36-17

- “Create a Custom Client (Host) Transport Protocol” on page 36-17
- “MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files” on page 36-18
- “Register a Custom Client (Host) Transport Protocol” on page 36-20
- “Create a Custom Server (Target) Transport Protocol” on page 36-21
- “Serial Receive Buffer Smaller than 64 Bytes” on page 36-23

Requirements for Custom Transport Layers

- By default, `ext_svr.c` and `updown.c` use `malloc` to allocate buffers in target memory for messages, data collection, and other purposes, although there is also an option to preallocate static memory. If your target uses another memory allocation scheme, you must modify these modules.
- The target is assumed to support both `int32_T` and `uint32_T` data types.

Create a Custom Client (Host) Transport Protocol

To implement the client (host) side of your low-level transport protocol,

- 1 Edit the template file `matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c` to replace low-level communication calls with your own communication calls.
 - a Copy and rename the file to `rtiostream_name.c` (replacing *name* with a name meaningful to you).
 - b Replace the functions `rtIOStreamOpen`, `rtIOStreamClose`, `rtIOStreamSend`, and `rtIOStreamRecv` with functions (of the same name) that call your low-level communication primitives. These functions are called from other external mode modules via `rtiostream_interface.c`. For more information, see “Communications rtiostream API” in the Embedded Coder documentation.
 - c Build your `rtiostream` implementation into a shared library that exports the `rtIOStreamOpen`, `rtIOStreamClose`, `rtIOStreamRecv` and `rtIOStreamSend` functions.
- 2 Build the customized MEX-file executable using the MATLAB `mex` function. See “MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files” on page 36-18 for examples of `mex` invocations.

Do not replace the existing `ext_comm` MEX-file if you want to preserve its existing function. Instead, use the `-output` option to name the resulting executable (for example, `mex -output ext_myrtiostream_comm ...` builds `ext_myrtiostream_comm.mexext`, on Windows platforms).

- 3 Register your new client transport layer with the Simulink software, so that the transport can be selected for a model using the **Interface** pane of the Configuration Parameters dialog box. For details, see “Register a Custom Client (Host) Transport Protocol” on page 36-20.

Sample commands for rebuilding external mode MEX-files are listed in “MATLAB Commands to Rebuild `ext_comm` and `ext_serial_win32` MEX-Files” on page 36-18.

MATLAB Commands to Rebuild `ext_comm` and `ext_serial_win32` MEX-Files

The following table lists the commands for building the standard `ext_comm` and `ext_serial_win32` modules on PC and UNIX platforms.

Platform	Commands
Windows, TCP/IP	<pre>>> cd (matlabroot) >> mex toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_comm.c ... toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_convert.c ... toolbox\coder\simulinkcoder_core\ext_mode\host\common\rtiostream_interface.c ... toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_util.c ... -Irtw\c\src -Irtw\c\src\rtiostream\utils ... -Irtw\c\src\ext_mode\common ... -Itoolbox\coder\simulinkcoder_core\ext_mode\host\common ... -Itoolbox\coder\simulinkcoder_core\ext_mode\host\common\include ... -lmwruntimeutils -lsl_services ... -DEXTMODE_TCPIP_TRANSPORT ... -DSL_EXT_DLL -output toolbox\coder\simulinkcoder_core\ext_comm</pre> <p>Note: The <code>rtiostream_interface.c</code> function defines <code>RTIOSTREAM_SHARED_LIB</code> as <code>libmwruntimeutils</code> and dynamically loads the MathWorks TCP/IP <code>rtiostream</code> shared library. Modify this file if you need to load a different <code>rtiostream</code> shared library.</p>
Linux, TCP/IP	<p>Use the Windows commands, with these changes:</p> <ul style="list-style-type: none"> • Change <code>-DSL_EXT_DLL</code> to <code>-DSL_EXT_S0</code>. • Change <code>-lsl_services</code> to <code>-lmwsl_services</code>. • Replace back slashes with forward slashes.

Platform	Commands
Mac, TCP/IP	<p>Use the Windows commands, with these changes:</p> <ul style="list-style-type: none"> • Change <code>-DSL_EXT_DLL</code> to <code>-DSL_EXT_DYLIB</code>. • Change <code>-lsl_services</code> to <code>-lmwsl_services</code>. • Replace back slashes with forward slashes.
Windows, serial	<pre data-bbox="348 489 1340 835">>> cd (matlabroot) >> mex toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_comm.c ... toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_convert.c ... toolbox\coder\simulinkcoder_core\ext_mode\host\serial\ext_serial_transport.c ... toolbox\coder\simulinkcoder_core\ext_mode\host\serial\ext_serial_pkt.c ... toolbox\coder\simulinkcoder_core\ext_mode\host\serial\rtiostream_serial_interface.c ... toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_util.c ... -Irtw\c\src -Irtw\c\src\rtiostream\utils ... -Irtw\c\src\ext_mode\common ... -Irtw\c\src\ext_mode\serial ... -Itoolbox\coder\simulinkcoder_core\ext_mode\host\common ... -Itoolbox\coder\simulinkcoder_core\ext_mode\host\common\include ... -lmwrtiostreamutils -lsl_services ... -DEXTMODE_SERIAL_TRANSPORT -DSL_EXT_DLL ... -output toolbox\coder\simulinkcoder_core\ext_serial_win32_comm</pre> <p data-bbox="348 892 1340 1020">Note: The <code>rtiostream_interface.c</code> function defines <code>RTIOSTREAM_SHARED_LIB</code> as <code>libmwrtiostreamserial</code> and dynamically loads the MathWorks serial <code>rtiostream</code> shared library. Modify this file if you need to load a different <code>rtiostream</code> shared library.</p>
Linux, serial	<p>Use the Windows commands, with these changes:</p> <ul style="list-style-type: none"> • Change <code>-DSL_EXT_DLL</code> to <code>-DSL_EXT_S0</code>. • Change <code>-lsl_services</code> to <code>-lmwsl_services</code>. • Replace back slashes with forward slashes.
Mac, serial	<p>Use the Windows commands, with these changes:</p> <ul style="list-style-type: none"> • Change <code>-DSL_EXT_DLL</code> to <code>-DSL_EXT_DYLIB</code>. • Change <code>-lsl_services</code> to <code>-lmwsl_services</code>. • Replace back slashes with forward slashes.

Note: `mex` requires a compiler supported by the MATLAB API. See the `mex` reference page and “MATLAB API for Other Languages” for more information about the `mex` function.

Register a Custom Client (Host) Transport Protocol

To register a custom client transport protocol with the Simulink software, you must add an entry of the following form to an `sl_customization.m` file on the MATLAB path:

```
function sl_customization(cm)
    cm.ExtModeTransports.add('stf.tlc', 'transport', 'mexfile', 'Level1');
% -- end of sl_customization
```

where

- `stf.tlc` is the name of the system target file for which the transport will be registered (for example, 'grt.tlc')
- `transport` is the transport name to display in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box (for example, 'mytcpip')
- `mexfile` is the name of the transport's associated external interface MEX-file (for example, 'ext_mytcpip_comm')

You can specify multiple targets and/or transports with additional `cm.ExtModeTransports.add` lines, for example:

```
function sl_customization(cm)
    cm.ExtModeTransports.add('grt.tlc', 'mytcpip', 'ext_mytcpip_comm', 'Level1');
    cm.ExtModeTransports.add('ert.tlc', 'mytcpip', 'ext_mytcpip_comm', 'Level1');
% -- end of sl_customization
```

If you place the `sl_customization.m` file containing the transport registration information on the MATLAB path, your custom client transport protocol will be registered with each subsequent Simulink session. The name of the transport will appear in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box. When you select the transport for your model, the name of the associated external interface MEX-file will appear in the noneditable **MEX-file name** field, as shown in the following figure.

External mode

External mode configuration

Transport layer: MEX-file name: ext_mytcpip_comm

MEX-file arguments:

Static memory allocation

Create a Custom Server (Target) Transport Protocol

The `rtIOStream*` function prototypes in `matlabroot/rtw/c/src/rtiostream.h` define the calling interface for both the server (target) and client (host) side transport layer functions.

- The TCP/IP implementations are in `matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c`.
- The serial implementations are in `matlabroot/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c`.

Note: The `Ext*` function prototypes in `matlabroot/rtw/c/src/ext_mode/common/ext_svr_transport.h` are implemented in `matlabroot/rtw/c/src/ext_mode/common/rtiostream_interface.c` or `matlabroot/rtw/c/src/ext_mode/serial/rtiostream_serial_interface.c`. In most cases you will not need to modify `rtiostream_interface.c` or `rtiostream_serial_interface.c` for your custom TCP/IP or serial transport layer.

To implement the server (target) side of your low-level TCP/IP or serial transport protocol:

- 1 Edit the template `matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c` or `matlabroot/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c` to replace low-level communication calls with your own communication calls.
 - a Copy and rename the file to `rtiostream_name.c` (replacing *name* with a name meaningful to you).

- b** Replace the functions `rtIOStreamOpen`, `rtIOStreamClose`, `rtIOStreamSend`, and `rtIOStreamRecv` with functions (of the same name) that call your low-level communication drivers.

You must implement the functions defined in `rtiostream.h`, and your implementations must conform to the prototypes defined in that file. Refer to the original `rtiostream_tcpip.c` or `rtiostream_serial.c` for guidance.

- 2** Incorporate the external mode source files for your transport layer into the model build process, according to your target type:

- If your target uses toolchain controls to configure a build, use a build process mechanism such as a post code generation command or a `before_make` hook function to make the transport files available to the build process. (For more information on the build process mechanisms, see “Customize Post-Code-Generation Build Processing” on page 35-14, “Customize Build Process with `STF_make_rtw_hook` File” on page 35-29, and “Customize Build Process with `sl_customization.m`” on page 35-36.) For example:

- Add the file created in the previous step to the build information:

```
path/rtiostream_name.c
```

- For TCP/IP, add the following file to the build information:

```
matlabroot/rtw/c/src/ext_mode/common/rtiostream_interface.c
```

- For serial, add the following files to the build information:

```
matlabroot/rtw/c/src/ext_mode/serial/ext_serial_pkt.c  
matlabroot/rtw/c/src/ext_mode/serial/rtiostream_serial_interface.c  
matlabroot/rtw/c/src/ext_mode/serial/ext_svr_serial_transport.c
```

- If your target uses template makefile controls to configure a build, modify template makefiles to support the new transport. Be sure to include the file created in the previous step, `rtiostream_name.c`. If you are writing your own template makefile, make sure that the `EXT_MODE` code generation option is defined. The generated makefile will then link `rtiostream_name.c`, `rtiostream_interface.c` or `rtiostream_serial_interface.c`, and other server code into your executable.

Note: For external mode, check that `rtIOStreamRecv` is not a blocking implementation. Otherwise, it might cause the external mode server to block until the host sends data through the `comm` layer.

Serial Receive Buffer Smaller than 64 Bytes

For serial communication, if the serial receive buffer of your target is smaller than 64 bytes:

- 1 Update the following macro with the actual target buffer size:

```
#define TARGET_SERIAL_RECEIVE_BUFFER_SIZE 64
```

Implement the change in the following files:

```
matlabroot/rtw/c/src/ext_mode/serial/ext_serial_utils.c  
matlabroot/toolbox/coder/simulinkcoder_core/ext_mode/host/serial/ext_serial_utils.c
```

- 2 Run the command to rebuild the `ext_serial_win32` MEX-file. See “MATLAB Commands to Rebuild `ext_comm` and `ext_serial_win32` MEX-Files” on page 36-18.

Custom Target Development in Simulink Coder

- “About Embedded Target Development” on page 37-2
- “Sample Custom Targets” on page 37-9
- “Target Development Folders, Files, and Builds” on page 37-11
- “Customize System Target Files” on page 37-29
- “Customize Template Makefiles” on page 37-62
- “Custom Target Optional Features” on page 37-79
- “Support Toolchain Approach with Custom Target” on page 37-81
- “Support Model Referencing” on page 37-83
- “Support Compiler Optimization Level Control” on page 37-95
- “Support C Function Prototype Control” on page 37-97
- “Support C++ Class Interface Control” on page 37-100
- “Support Concurrent Execution of Multiple Tasks” on page 37-102
- “Interface to Development Tools” on page 37-104
- “Device Drivers” on page 37-116

About Embedded Target Development

The system target files bundled with the code generator are suitable for many different applications and development environments. Third-party targets provide additional versatility. If necessary, you can implement a custom target.

To implement a target based on the ARM[®] Cortex[®]-A or ARM Cortex-M processor, install the corresponding support package and see the Target SDK: Embedded Coder Support Package for ARM Cortex-A Processors, “Develop a Target” or Embedded Coder Support Package for ARM Cortex-M Processors, “Develop a Target”. Otherwise, use these functions and topics.

Custom Targets

You might want to implement a custom target for one of the following reasons:

- To enable end users to generate executable production code for a specific CPU or development board, using a specific development environment (compiler/linker/debugger).
- To support I/O devices on the target hardware by incorporating custom device driver blocks into your models.
- To configure the build process for a special compiler (such as a cross-compiler for an embedded microcontroller or DSP board) or development/debugging environment.

The code generator provides a point of departure for the creation of custom embedded targets, for the basic purposes above. This manual covers the tasks and techniques you need to implement a custom embedded target.

Types of Targets

The following sections describe several types of targets intended for different use cases

- “About Target Types” on page 37-3
- “Rapid Prototyping Targets” on page 37-3
- “Turnkey Production Targets” on page 37-3
- “Verifying Targets With SIL and PIL Testing” on page 37-4
- “HIL Simulation Targets” on page 37-4

About Target Types

There is a progression of capabilities from the first (baseline or rapid prototyping) to second (production) target types; you may want to implement an initial rapid prototyping target and a following, more full-featured turnkey version of a target. You might want to use software-in-the-loop (SIL) or processor-in-the-loop (PIL) testing at a particular stage to verify your embedded target. The target types are not mutually exclusive. An embedded target can support more than one of these use cases, or additional uses not outlined here.

The discussion of target types is followed by “Recommended Features for Embedded Targets” on page 37-4, which contains a suggested list of target features and general guidelines for embedded target development.

Rapid Prototyping Targets

A *rapid prototyping target* or baseline target offers a starting point for targeting a production processor. A rapid prototyping target integrates coded generator software with one or more popular cross-development environments (compiler/linker/debugger tool chains). A rapid prototyping target provides a starting point from which you can customize the target for application needs.

Target files provided for this type of target should be readable, easy to understand, and fully commented and documented. Specific attention should be paid to the interface to the intended cross-development environment. This interface should be implemented using the preferred approach for that particular development system. For example, some development environments use traditional make utilities, while others are based on project-file builds that can be automated under control of the code generator.

When you use a rapid prototyping target, you need to include your own device driver and legacy code and modify linker memory maps to suit your needs. You should be familiar with the targeted development system.

Turnkey Production Targets

A *turnkey production target* also targets a production processor, but includes the capability to create target executables that interact immediately with the external world. In general, ease of use is more important than simplicity or readability of the target files, because it is assumed that you do not want or need to modify these files.

Desirable features for a turnkey production target include

- Significant I/O driver support provided out of the box
- Easy downloading of generated standalone executables with third-party debuggers
- User-controlled placement of an executable in FLASH or RAM memory
- Support for target visibility and tuning

Verifying Targets With SIL and PIL Testing

You can use software-in-the-loop (SIL) or processor-in-the-loop (PIL) to verify your generated code and validate the target compiler/processor environment.

You can use SIL and PIL simulation mode to verify automatically generated code by comparing the results with a normal mode simulation. With SIL, you can easily verify the behavior of production-intent source code on your host computer; however, it is generally not possible to verify exactly the same code that will subsequently be compiled for your target hardware because the code must be compiled for your host platform (i.e. a different compiler and different processor architecture than the target). With PIL simulation, you can verify exactly the same code that you intend to deploy in production, and you can run the code either on real target hardware or on an instruction set simulator.

For examples describing how to run processor-in-the-loop testing to verify a custom target, see “Sample Custom Targets” on page 37-9.

For more information on SIL and PIL, see “SIL and PIL Simulations”.

HIL Simulation Targets

A specialized use case is the generation of executables intended for use in *hardware-in-the-loop* (HIL) simulations. In a HIL simulation, parts of a pure simulation are gradually replaced with hardware components as components are refined and fabricated. HIL simulation offers an efficient design process that eliminates costly iterations of part fabrication.

Recommended Features for Embedded Targets

- “Basic Target Features” on page 37-5
- “Integration with Target Development Environments” on page 37-6
- “Observing Execution of Target Code” on page 37-6
- “Deployment and Hardware Issues” on page 37-7

Basic Target Features

- You can base targets on the generic real-time (GRT) target or the Embedded Real-Time (ERT) target that is included in the Embedded Coder product.

If your target is based on the ERT target, it should use ‘Embedded-C’ value for the `CodeFormat` TLC variable, and it should inherit the options defined in the ERT target's system target file with the following lines in the TLC file:

```
%% Assign code format
%assign CodeFormat = "Embedded-C"
%%-----
/%
  BEGIN_RTW_OPTIONS
  rtwgensettings.DerivedFrom = 'ert.tlc';
  END_RTW_OPTIONS
%/
%%-----
```

By following these recommendations, your target has the production code generation capabilities of the ERT target.

See “Customize System Target Files” on page 37-29 for further details on the inheritance mechanism, setting the `CodeFormat`, and other details.

- The most fundamental requirement for an embedded target is that it generate a real-time executable from a model or subsystem. Typically, an embedded target generates a timer interrupt-based, bareboard executable (although targets can be developed for an operating system environment as well).

Your target should support code generator concepts of single-tasking and multitasking solver modes for model execution. Tasking support is available with the ERT target, but you should thoroughly understand how it works before implementing an ERT-based target.

For information on timer interrupt-based execution, see “Absolute and Elapsed Time Computation” on page 3-2 and “Asynchronous Events” on page 5-2.

- You should generate the target executable's main program module, rather than using a static main module (such as the static `rt_main.c` or `rt_cppclass_main.cpp` module provided with the software). A generated `main.c` or `.cpp` can be made much more readable and more efficient, since it omits preprocessor checks and other extra code.

See “Deploy Generated Standalone Executables To Target Hardware” for information on generated and static main program modules.

- Follow the guidelines in “Folder and File Naming Conventions” on page 37-11.

Integration with Target Development Environments

- Most cross-development systems run under a Microsoft Windows PC host. Your target should support the Windows operating system as the host environment.

Some cross-development systems support one or more versions of The Open Group UNIX platforms, allowing for UNIX host support as well.

- Your embedded target must support at least one embedded development environment. The interface to a development environment can take one of several forms. The toolchain approach and template makefile approach generate standard makefiles to work with your development environment. For general information about these build approaches, see “Configure Build Process” on page 25-13. For detailed information about the structure of template makefiles, see “Customize Template Makefiles” on page 37-62.

Another approach with IDE-based tools is to create a Microsoft Visual Studio® Solution from your target for integration within a Visual Studio project.

It is important to consider the license requirements and restrictions of the development environment vendor. You may need to modify files provided by the vendor and ship them as part of the embedded target.

See “Interface to Development Tools” on page 37-104 for further information.

Observing Execution of Target Code

- Your target should support a mechanism you can use to observe the target code as it runs in real time (outside of a debugger).

You can use the `rtiostream` API to implement a communication channel to enable exchange of data between different processes. For an example of creating a communication channel for target connectivity, see “Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation”. This `rtiostream` communication channel is required to enable processor-in-the-loop (PIL) on a new target. See “Communications `rtiostream` API” in the Embedded Coder documentation.

One industry-standard approach is to use the CAN bus, with an ASAP2 file and CAN Calibration Protocol (CCP). There are several host-based graphical front-end tools available that connect to a CCP-enabled target and provide data viewing and parameter tuning. Supporting these tools requires implementation of CAN hardware drivers and CCP protocol for the target, as well as ASAP2 file generation. Your target can leverage the ASAP2 support provided with the code generator.

Another option is to support Simulink external mode over a serial interface (RS-232). See the “What You Can Do With a Host/Target Communication Channel” on page 26-2 for information on using the external mode API.

Deployment and Hardware Issues

- Device driver support is an important issue in the design of an embedded target. Device drivers are Simulink blocks that support either hardware I/O capabilities of the target CPU, or I/O features of the development board.

If you are developing a rapid prototyping target, consider providing minimal driver support, on the assumption that end users develop their own drivers. If you are developing a turnkey production target, you should provide full driver support.

See “Device Drivers” on page 37-116.

- Automatic download of generated code to the target hardware makes a target easier to use. Typically a debugger utility is used; if the chosen debugger supports command script files, this can be straightforward to implement. “STF_make_rtw_hook.m” on page 37-21 describes a mechanism to execute code from the build process. You can use this mechanism to make `system()` calls to invoke utilities such as a debugger. You can invoke other simple downloading utilities in a similar fashion.

If your development system supports COM automation, you can control the download process by that mechanism. Using COM automation is discussed in “Interface to Development Tools” on page 37-104.

- Executables that are mapped to RAM memory are typical. You can provide optional support for FLASH or RAM placement of the executable by using your target's code generation options. To support this capability, you might need multiple linker command files, multiple debugger scripts, and possibly multiple makefiles or project files. Also include the ability to automatically switch between these files, depending on the RAM/FLASH option value.

- Select a popular, widely available evaluation or prototype board for your target processor. Consider enclosed and ruggedized versions of the target board. Also consider board level support for the various on-chip I/O capabilities of the target CPU, and the availability of development systems that support the selected board.

More About

- “Sample Custom Targets” on page 37-9
- “Target Development Folders, Files, and Builds” on page 37-11
- “Customize System Target Files” on page 37-29
- “Customize Template Makefiles” on page 37-62
- “Custom Target Optional Features” on page 37-79
- “Support Toolchain Approach with Custom Target” on page 37-81
- “Support Model Referencing” on page 37-83
- “Support Compiler Optimization Level Control” on page 37-95
- “Support C Function Prototype Control” on page 37-97
- “Support C++ Class Interface Control” on page 37-100
- “Support Concurrent Execution of Multiple Tasks” on page 37-102
- “Interface to Development Tools” on page 37-104
- “Device Drivers” on page 37-116

Sample Custom Targets

There are technical solutions on the MathWorks Web site that you can use as a starting point to create your own target solution. The solutions provide guides to the following tasks for creating custom targets:

- Methods of embedding code onto a custom processor
- Creating a system target file
- Customizing the makefile and main file
- Adding compiler, chip, and board specific information
- Integrating legacy code and device drivers
- Creating blocks and libraries
- Implementing processor-in-the-loop (PIL) testing.

- 1 Start by downloading the embedded targets development guide zip file from this web page:

Is there an example guide on developing an embedded target...?

The zip file provides example files and a guide to developing a custom embedded target. The guide is divided into two parts, one on creating a generic custom target and another on creating a target for the Freescale™ S12X processor using the Cosmic Compiler.

Read the example guide along with this document to understand the tasks for developing embedded targets.

- 2 For more detailed example files for specific processors, see:

- Is there an example Freescale S12X target... using the Cosmic Compiler?
- Is there an example Freescale S12X target... using the CodeWarrior Compiler?

These example kits contain example models, code generation files, and instruction guides on generating and testing code for the processor. The Cosmic example illustrates the use of the target connectivity API for processor-in-the-loop (PIL) testing. The CodeWarrior example does not have PIL but shows CAN Calibration Protocol (CCP) and Simulink external mode.

The intent of the example kits is to provide working examples that you can use as a base to create your own target solution. The intent is not to provide a full featured

and maintained Embedded Target product like those provided by MathWorks or third-party products, as listed on the Embedded Coder Hardware Support Web page.

3 You can watch videos showing overviews of both the example kits at the following links:

- www.mathworks.com/videos/programming-the-freescale-s12x-target-68811.html
- www.mathworks.com/videos/programming-arm9-using-the-hitex-str9-comstick-68812.html

For another example target for the ARM9 (STR9) processor, see [Is there an example ARM9 \(STR9\) target... using the GNU ARM Compiler and Hitex STR9-comStick?](#).

If you have questions on specific targets, please email mytarget@mathworks.com.

The example kits and this document describe Embedded Coder features such as customized ert system target files and processor-in-the-loop testing, but you can study the examples as a starting point for use with Simulink Coder targets.

More About

- “About Embedded Target Development” on page 37-2
- “Customize System Target Files” on page 37-29
- “Customize Template Makefiles” on page 37-62
- “Custom Target Optional Features” on page 37-79
- “Support Toolchain Approach with Custom Target” on page 37-81
- “Interface to Development Tools” on page 37-104
- “Device Drivers” on page 37-116

Target Development Folders, Files, and Builds

Target development mechanics work with a number of folder and file types. The following topics provide the information needed to develop custom targets, configure folder usage, and use custom targets in the build process.

In this section...

“Folder and File Naming Conventions” on page 37-11

“Components of a Custom Target” on page 37-12

“Key Folders Under Target Root (mytarget)” on page 37-17

“Key Files in Target Folder (mytarget/mytarget)” on page 37-19

“Additional Files for Externally Developed Targets” on page 37-22

“Target Development and the Build Process” on page 37-23

Folder and File Naming Conventions

You can use a single folder for your custom target files, or if desired you can use subfolders, for example containing files associated with specific development environments or tools.

For a custom target implementation, the recommended folder and file naming conventions are

- Use *only* lowercase in folder names, filenames, and extensions.
- Do not embed spaces in folder names. Spaces in folder names cause errors with many third-party development environments.
- Include desired folders in the MATLAB path
- Do *not* place your custom target folder anywhere in the MATLAB folder tree (that is, in or under the *matlabroot* folder). If you place your folder under *matlabroot* you risk losing your work if you install a new MATLAB version (or reinstall the current version).

The following sections explain how to organize your target folders and files and add them to the your MATLAB path. They also provide high-level descriptions of the files.

In this document, *mytarget* is a placeholder name that represents folders and files that use the target's name. The names *dev_tool1*, *dev_tool2*, and so on represent

subfolders containing files associated with development environments or tools. This document describes an example structure where the folder `mytarget` contains subfolders for `mytarget`, `blocks`, `dev_tool1`, `dev_tool2`. The top level folder `mytarget` is the *target root folder*.

Components of a Custom Target

- “Overview” on page 37-12
- “Code Components” on page 37-13
- “Control Files” on page 37-14

Overview

The components of a custom target are files located in a hierarchy of folders. The top-level folder in this structure is called the *target root folder*. The target root folder and its contents are named, organized, and located on the MATLAB path according to conventions described in “Folder and File Naming Conventions” on page 37-11.

The components of a custom target include

- Code components: C source code that supervises and supports execution of generated model code.
- Control files:
 - A system target file (STF) to control the code generation process.
 - File(s) to control the building of an executable from the generated code. In a traditional make-based environment, a template makefile (TMF) generates a makefile for this purpose. Another approach is to generate project files in support of a modern integrated development environment (IDE) such as the Freescale Semiconductor CodeWarrior IDE.
 - Hook files: Optional TLC and MATLAB program files that can be invoked at well-defined stages of the build process. Hook files let you customize the build process and communicate information between various phases of the process.
- Other target files: Files that let you integrate your target into the MATLAB environment. For example, you can provide an `info.xml` file to make your target block libraries and examples available from a MATLAB session.

The next sections introduce key concepts and terminology you need to know to develop each component. References to more detailed information sources are provided.

Code Components

An executable program containing code generated from a Simulink model consists of a number of code modules and data structures. These fall into two categories.

Application Components

Application components are those which are specific to a particular model; they implement the functions represented by the blocks in the model. Application components are not specific to the target. Application components include

- Modules generated from the model
- User-written blocks (S-functions)
- Parameters of the model that are visible, and can be interfaced to, external code

Execution Support Files

A number of code modules and data structures, referred to collectively as the *execution support files*, are responsible for managing and supporting the execution of the generated program. The execution support files modules are not automatically generated. Depending on the requirements of your target, you must implement certain parts of the execution support files. Execution Support Files summarizes the execution support files.

Execution Support Files

You Provide...	The Code Generator Provides...
Customized main program	Generic main program
Timer interrupt handler to run model	Execution engine and integration solver (called by timer interrupt handler)
Other interrupt handlers	Example interrupt handlers (Asynchronous Interrupt blocks)
Device drivers	Example device drivers
Data logging, parameter tuning, signal monitoring, and external mode support	Data logging, parameter tuning, signal monitoring, and external mode APIs

User-Written Execution Support Files

The code generator provides most of the execution support files. Depending on the requirements of your target, you must implement some or all of the following elements:

- A timer *interrupt service routine* (ISR). The timer runs at the program's base sample rate. The timer ISR is responsible for operations that must be completed within a single clock period, such as computing the current output sample. The timer ISR usually calls the `rt_OneStep` function.

If you are targeting a real-time operating system (RTOS), your generated code usually executes under control of the timing and task management mechanisms provided by the RTOS. In this case, you may not have to implement a timer ISR.

- The *main program*. Your main program initializes the blocks in the model, installs the timer ISR, and executes a background task or loop. The timer periodically interrupts the main loop. If the main program is designed to run for a finite amount of time, it is also responsible for cleanup operations — such as memory deallocation and masking the timer interrupt — before terminating the program.

If you are targeting a real-time operating system (RTOS), your main program most likely spawns tasks (corresponding to the sample rates used in the model) whose execution is timed and controlled by the RTOS.

Your main program typically is based on a generated or static main program. “Deploy Generated Standalone Executables To Target Hardware” details the structure of the execution support files and the execution of code, and provides guidelines for customizing main programs.

- *Device drivers*. Drivers communicate with I/O devices on your target hardware. In production code, device drivers are normally implemented as inlined S-functions.
- *Other interrupt handlers*. If your models need to support asynchronous events, such as hardware generated interrupts and asynchronous read and write operations, you must supply interrupt handlers. The Interrupt Templates library provides examples.
- *Data logging, parameter tuning, signal monitoring, and external mode support*. It is atypical to implement rapid prototyping features such as external mode support in an embedded target. However, it is possible to support these features by using standard code generator APIs. See “Data Exchange Interfaces” for details.

Control Files

The code generation and build process is directed by a number of TLC and MATLAB files collectively called *control files*. This section introduces and summarizes the main control files.

Top-Level Control File (`make_rtw`)

The build process is initiated when you press **Ctrl+B**. At this point, the build process parses the **Make command** field of the **Code Generation** pane of the Configuration Parameters dialog box, expecting to find the name of a MATLAB command that controls the build process (as well as optional arguments to that command). The default command is `make_rtw`, and the default top-level control file for the build process is `make_rtw.m`.

Note: `make_rtw` is an internal MATLAB command used by the build process. Normally, target developers do not need detailed knowledge of how `make_rtw` works. (The details for target developers are described in “Target Development and the Build Process” on page 37-23.) You should not invoke `make_rtw` directly from MATLAB code, and you should not customize `make_rtw.m`.

The `make_rtw.m` file contains the logic required to execute your target-specific control files, including a number of hook points for execution of your custom code. `make_rtw` does the following:

- Passes optional arguments in to the build process
- Performs required preprocessing before code generation
- Executes the STF to perform code generation (and optional HTML report generation)
- Processes the TMF to generate a makefile
- Invokes a make utility to execute the makefile and build an executable
- Performs required post-processing (such as generating calibration data files or downloading the generated executable to the target)

System Target File (STF)

The Target Language Compiler (TLC) generates target-specific C or C++ code from an partial description of your Simulink block diagram (`model.rtw`). The Target Language Compiler reads `model.rtw` and executes a program consisting of several target files (`.tlc` files.) The STF, at the top level of this program, controls the code generation process. The output of this process is a number of source files, which are fed to your development system's make utility.

You need to create a customized STF to set code generation parameters for your target. You should copy, rename, and modify the standard ERT system target file (`matlabroot/rtw/c/ert/ert.tlc`).

The detailed structure of the STF is described in “Customize System Target Files” on page 37-29.

Note: The STF selects whether the target supports the toolchain approach or template makefile approach for code generation. See “Customize System Target Files” on page 37-29.

Template Makefile (TMF)

A TMF provides information about your model and your development system. The build process uses this information to create a makefile (.mk file) that builds an executable program.

Some targets implement more than one TMF, in order to support multiple development environments (for example, two or more cross-compilers) or multiple modes of code generation (for example, generating a binary executable versus generating a project file for your compiler).

The Embedded Coder software provides a large number of TMFs suitable for different types of development computer systems. These TMFs are located in *matlabroot*/rtw/c/ert (open). The standard TMFs are described in “Template Makefiles and Make Options” on page 17-26.

The detailed structure of the TMF is described in “Customize Template Makefiles” on page 37-62.

Note: The STF selects whether the target supports the toolchain approach or template makefile approach for code generation. See “Customize System Target Files” on page 37-29.

Hook Files

The build process allows you to supply optional *hook files* that are executed at specified points in the code generation and make process. You can use hook files to add target-specific actions to the build process.

The hook files must follow well-defined naming and location requirements. “Folder and File Naming Conventions” on page 37-11 describes these requirements.

Key Folders Under Target Root (mytarget)

- “Target Root Folder (mytarget)” on page 37-17
- “Target Folder (mytarget/mytarget)” on page 37-17
- “Target Block Folder (mytarget/blocks)” on page 37-17
- “Development Tools Folder (mytarget/dev_tool1, mytarget/dev_tool2)” on page 37-19
- “Target Source Code Folder (mytarget/src)” on page 37-19

Target Root Folder (mytarget)

This folder contains the key subfolders for the target (see “Folder and File Naming Conventions” on page 37-11). You can also locate miscellaneous files (such as a `readme` file) in the target root folder. The following sections describe required and optional subfolders and their contents.

Target Folder (mytarget/mytarget)

This folder contains files that are central to the target, such as the system target file (STF) and template makefile (TMF). “Key Files in Target Folder (mytarget/mytarget)” on page 37-19 summarizes the files that should be stored in `mytarget/mytarget`, and provides pointers to detailed information about these files.

Note `mytarget/mytarget` should be on the MATLAB path.

Target Block Folder (mytarget/blocks)

If your target includes device drivers or other blocks, locate the block implementation files in this folder. `mytarget/blocks` contains

- Compiled block MEX-files
- Source code for the blocks
- TLC inlining files for the blocks
- Library models for the blocks (if you provide your blocks in one or more libraries)

Note `mytarget/blocks` should be on the MATLAB path.

You can also store example models and supporting files in `mytarget/blocks`. Alternatively, you can create a `mytarget/mytargetdemos` folder, which should also be on the MATLAB path.

To display your blocks in the standard Simulink Library Browser and/or integrate your example models into the MATLAB session environment, you can create the files described below and store them in `mytarget/blocks`.

mytarget/blocks/siblocks.m

This file allows a group of blocks to be integrated into the Simulink Library and Simulink Library Browser.

Example siblocks.m File

```
function blkStruct = siblocks
% Information for "Blocksets and Toolboxes" subsystem
blkStruct.Name = sprintf('Embedded Target\n for MYTARGET');
blkStruct.OpenFcn = 'mytargetlib';
blkStruct.MaskDisplay = 'disp(''MYTARGET'')';

% Information for Simulink Library Browser
Browser(1).Library = 'mytargetlib';
Browser(1).Name = 'Embedded Target for MYTARGET';
Browser(1).IsFlat = 1;% Is this library "flat" (i.e. no subsystems)?

blkStruct.Browser = Browser;
```

mytarget/blocks/demos.xml

This file provides information about the components, organization, and location of example models. MATLAB software uses this information to place the example in the MATLAB session environment.

Example demos.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<demos>
  <name>Embedded Target for MYTARGET</name>
  <type>simulink</type>
  <icon>$toolbox/matlab/icons/boardicon.gif</icon>
  <description source = "file">mytarget_overview.html</description>

  <demosession>
    <label>Multirate model</label>
    <demositem>
      <label>MYTARGET demo</label>
      <file>mytarget_overview.html</file>
      <callback>mytarget_model</callback>
```



```
</demoitem>  
</demosession>  
  
</demos>
```

Development Tools Folder (mytarget/dev_tool1, mytarget/dev_tool2)

These folders contain files associated with specific development environments or tools (`dev_tool1`, `dev_tool2`, etc.). Normally, your target supports at least one such development environment and invokes its compiler, linker, and other utilities during the build process. `mytarget/dev_tool1` includes linker command files, startup code, hook functions, and other files required to support this process.

For each development environment, you should provide a separate folder.

Target Source Code Folder (mytarget/src)

This folder is optional. If the complexity of your target requires it, you can use `mytarget/src` to store common source code and configuration code (such as boot and startup code).

Key Files in Target Folder (mytarget/mytarget)

- “Introduction” on page 37-19
- “mytarget.tlc” on page 37-20
- “mytarget.tmf” on page 37-20
- “mytarget_genfiles.tlc” on page 37-20
- “mytarget_main.c” on page 37-20
- “STF_make_rtw_hook.m” on page 37-21
- “STF_rtw_info_hook.m (obsolete)” on page 37-21
- “info.xml” on page 37-21
- “mytarget_overview.html” on page 37-21

Introduction

The target folder `mytarget/mytarget` contains key files in your target implementation. These include the system target file, template makefile, main program module, and optional M and TLC hook files that let you add target-specific actions to the build process. The following sections describe the key target folder files.

mytarget.tlc

`mytarget.tlc` is the system target file (STF). Functions of the STF include

- Making the target visible in the System Target File Browser
- Definition of code generation options for the target (inherited and target-specific)
- Providing an entry point for the top-level control of the TLC code generation process

You should base your STF on `ert.tlc`, the STF provided by Embedded Coder software.

“Customize System Target Files” on page 37-29 gives detailed information on the structure of the STF, and also gives instructions on how to customize an STF to

- Display your target in the System Target File Browser
- Add your own target options to the Configuration Parameters dialog box
- Tailor the code generation and build process to the requirements of your target

mytarget.tmf

`mytarget.tmf` is the template makefile for building an executable for your target.

For basic information on the structure and operation of template makefiles, see “Customize Template Makefiles” on page 37-62.

If your target development environment requires automation of a modern integrated development environment (IDE) rather than use of a traditional make utility, see “Interface to Development Tools” on page 37-104.

mytarget_genfiles.tlc

This file is optional. `mytarget_genfiles.tlc` is useful as a central file from which to invoke target-specific TLC files that generate additional files as part of your target build process. For example, your target may create sub-makefiles or project files for a development environment, or command scripts for a debugger to do automatic downloads. See “Using `mytarget_genfiles.tlc`” on page 37-45 for details.

mytarget_main.c

A main program module is required for your target. To provide a main module, you can either

- Modify the `rt_main.c` or `rt_cppclass_main.cpp` module provided by the software

- Generate `mytarget_main.c` or `.cpp` during the build process

“Deploy Generated Standalone Executables To Target Hardware” contains a detailed description of the operation of main programs. The section also contains guidelines for generating and modifying a main program module.

STF_make_rtw_hook.m

`STF_make_rtw_hook.m` is an optional hook file that you can use to invoke target-specific functions or executables at specified points in the build process. `STF_make_rtw_hook.m` implements a function that dispatches to a specific action depending on the `method` argument that is passed into it.

“Customize Build Process with `STF_make_rtw_hook` File” on page 35-29 describes the operation of the `STF_make_rtw_hook.m` hook file in detail.

STF_rtw_info_hook.m (obsolete)

Prior to Release 14, custom targets supplied target-specific information with a hook file (referred to as `STF_rtw_info_hook.m`). The `STF_rtw_info_hook` specified properties such as word sizes for integer data types (for example, `char`, `short`, `int`, and `long`), and C implementation-specific properties of the custom target.

The `STF_rtw_info_hook` mechanism has been replaced by the **Hardware Implementation** pane of the Configuration Parameters dialog box. Using this dialog box, you can specify the properties that were formerly specified in your `STF_rtw_info_hook` file.

For backward compatibility, existing `STF_rtw_info_hook` files are still available. However, you should convert your target and models to use the **Hardware Implementation** pane. See “Platform Options for Development and Deployment” on page 10-2.

info.xml

This file provides information to MATLAB software that specifies where to display the target toolbox in the MATLAB session environment. For more information, see “Display Custom Documentation” in the MATLAB documentation.

mytarget_overview.html

By convention, this file serves as home page for the target examples.

The `<description>` field in `demos.xml` should point to `mytarget_overview.html` (see “`mytarget/blocks/demos.xml`” on page 37-18).

Example `mytarget_overview.html` File

```
<html>
<head><title>Embedded Target for MYTARGET</title></head><body>
<p style="color:#990000; font-weight:bold; font-size:x-large">Embedded Target
for MYTARGET Example Model</p>

<p>This example provides a simple model that allows you to generate an executable
for a supported target board. You can then download and run the executable and
set breakpoints to study and monitor the execution behavior.</p>

</body>
</html>
```

Additional Files for Externally Developed Targets

- “Introduction” on page 37-22
- “`mytarget/mytarget/mytarget_setup.m`” on page 37-22
- “`mytarget/mytarget/doc`” on page 37-23

Introduction

If you are developing an embedded target that is not installed into the MATLAB tree, you should provide a target setup script and target documentation within `mytarget/mytarget`, for the convenience of your users. The following sections describe the required materials and where to place them.

`mytarget/mytarget/mytarget_setup.m`

This file script adds paths for your target to the MATLAB path. Your documentation should instruct users to run the script when installing the target.

You should include a call to the MATLAB function `savepath` in your `mytarget_setup.m` script. This function saves the added paths, so users need to run `mytarget_setup.m` only once.

The following code is an example `mytarget_setup.m` file.

```
function mytarget_setup()
curpath = pwd;
tgtpath = curpath(1:end-length('\mytarget'));
```

```
addpath(fullfile(tgtpath, 'mytarget'));
addpath(fullfile(tgtpath, 'dev_tool1'));
addpath(fullfile(tgtpath, 'blocks'));
addpath(fullfile(tgtpath, 'mytargetdemos'));
savepath;
disp('MYTARGET Target Path Setup Complete.');
```

mytarget/mytarget/doc

You should put the documentation related to your target in the folder `mytarget/mytarget/doc`.

Target Development and the Build Process

- “About the Build Process” on page 37-23
- “Build Process Phases and Information Passing” on page 37-23
- “Additional Information Passing Techniques” on page 37-26

About the Build Process

To develop an embedded target, you need a thorough understanding of the build process. Your embedded target uses the build process and may require you to modify or customize the process. A general overview of code generation and the build process is given in “Code Generation” and “Build Process”.

This section supplements that overview with a description of the build process as customized by the Embedded Coder software. The emphasis is on points in the process where customization hooks are available and on passing information between different phases of the process.

This section concludes with “Additional Information Passing Techniques” on page 37-26, describing assorted tips and tricks for passing information during the build process.

Build Process Phases and Information Passing

It is important to understand where (and when) the build process obtains required information. Sources of information include

- The `model.rtw` file, which provides information about the generating model. The information in `model.rtw` is available to target TLC files.

- The code generation panes of the Configuration Parameters dialog box. Options (both general and target-specific) are provided through check boxes, menus, and edit fields. You can associate options with TLC variables in the `rtwoptions` data structure. Use the **Configuration Parameters > Code Generation > Custom Code > Additional build information > Defines** field to define makefile tokens .
- The selected toolchain (for toolchain approach builds) or selected template makefile `.tmf` (for template makefile approach builds); these generate the model-specific makefile.
- Environment variables on the host computer. Environment variables provide additional information about installed development tools.
- Other target-specific files such as target-related TLC files, linker command files, or project files.

It is also important to understand the several phases of the build process and how to pass information between the phases. The build process comprises several high-level phases:

- Execution of the top-level file (`slbuild.m` or `rtwbuild.m`) to sequence through the build process for a target
- Conversion of the model into the TLC input file (`model.rtw`)
- Generation of the target code by the TLC compiler
- Compilation of the generated code with `make` or other utilities
- Transmission of the final generated executable to the target hardware with a debugger or download utility

It is helpful to think of each phase of the process as a different “environment” that maintains its own data. These environments include

- MATLAB code execution environment (MATLAB)
- Simulink
- Target Language Compiler execution environment
- makefile
- Development environments such as and IDE or debugger

In each environment, you might get information from the various sources mentioned above. For example, during the TLC phase, execute MATLAB file might execute to obtain information from the MATLAB environment. Also, a given phase may generate information for a subsequent phase.

See “Key Files in Target Folder (mytarget/mytarget)” on page 37-19 for details on the available MATLAB file and TLC hooks for information passing, with code examples.

Additional Information Passing Techniques

This section describes a number of useful techniques for passing information among different phases of the build process.

tlcvariable Field in rtwoptions Structure

Parameters on the code generation panes of the Configuration Parameters dialog box can be associated with a TLC variable, and specified in the `tlcvariable` field of the option's entry in the `rtwoptions` structure. The variable value is passed on the command line when TLC is invoked. This provides a way to make code generation parameters and their values available in the TLC phase.

See “System Target File Structure” on page 37-30 for further information.

makevariable Field in rtwoptions Structure

You can associate code generation parameters with a template makefile token, that you specify in the `makevariable` field of the option's entry in the `rtwoptions` structure. If a token of the same name as the `makevariable` name exists in the TMF, the token is updated with the option value when the final makefile is created. If the token does not exist in the TMF, the `makevariable` is passed in on the command line when `make` is invoked. Thus, in either case, the `makevariable` is available to the makefile.

See “System Target File Structure” on page 37-30 for further information.

Accessing Host Environment Variables

You can access host shell environment variables at the MATLAB command line by entering the `getenv` command. For example:

```
getenv ('MSDEVDIR')
```

```
ans =
```

```
D:\Applications\Microsoft Visual Studio\Common\MSDev98
```

To access the same information from TLC, use the `FEVAL` directive to invoke `getenv`.

```
%assign eVar = FEVAL("getenv", "<varname>")
```

Supplying Development Environment Information to Your Template Makefile

An embedded target must tie the build process to target-specific development tools installed on a host computer. For the make process to run these tools, the TMF must

be able to determine the name of the tools, the path to the compiler, linker, and other utilities, and possibly the host operating system environment variable settings.

Require the end user to modify the target TMF. The user enters path information (such as the location of a compiler executable), and possibly host operating system environment variables, as make variables. This allows the TMF to be tailored to specific needs.

Using MATLAB Application Data

Application data provides a way for applications to save and retrieve data stored with the GUI. This technique enables you to create what is essentially a user-defined property for an object, and use this property to store data for use in the build process. If you are unfamiliar with this technique, see the “Store Data as Application Data” section of the MATLAB documentation of creating graphical user interfaces.

The following code examples illustrates the use of application data to pass information to TLC.

This file, `tlc2appdata.m`, stores the data passed in as application data under the name passed in (`appDataName`).

```
function k = tlc2appdata(appDataName,data)
    disp([mfilename,': ',appDataName,' ', data]);
    setappdata(0,appDataName,data);
    k = 0; % TLC expects a return value for FEVAL.
```

The following sample TLC file uses the FEVAL directive to invoke `tlc2appdata.m` to store arbitrary application data, under the name `z80`.

```
%% test.tlc
%%
%assign myApp = "z80"
%assign myData = "314159"
%assign dummy = FEVAL("tlc2appdata",myApp,myData)
```

To test this technique:

- 1 Create the `tlc2appdata.m` file as shown. Check that `tlc2appdata.m` is stored in a folder on the MATLAB path.
- 2 Create the TLC file as shown. Save it as `test.tlc`.
- 3 Enter the following command at the MATLAB prompt to execute the TLC file:

```
tlc test.tlc
```

- 4 Get the application data at the MATLAB prompt:

```
k = getappdata(0, 'z80')
```

The function returns the value 314159.

- 5 Enter the following command.

```
who
```

Note that application data is not stored in the MATLAB workspace. Also observe that the z80 data is not visible. Using application data in this way has the advantage that it does not clutter the MATLAB workspace. Also, it helps prevent you from accidentally deleting your data, since it is not stored directly in the your workspace.

A real-world use of application data might be to collect information from the *model.rtw* file and store it for use later in the build process.

Adding Block-Specific Information to the Makefile

The `rtwmakecfg` mechanism provides a method for inlined S-functions such as driver blocks to add information to the makefile. This mechanism is described in “Use `rtwmakecfg.m` API to Customize Generated Makefiles” on page 35-24.

More About

- “About Embedded Target Development” on page 37-2

Customize System Target Files

This section provides information on the structure of the STF, guidelines for customizing an STF, and a basic tutorial that helps you get a skeletal STF up and running.

In this section...

“Control Code Generation With the System Target File” on page 37-29

“System Target File Naming and Location Conventions” on page 37-30

“System Target File Structure” on page 37-30

“Define and Display Custom Target Options” on page 37-37

“Tips and Techniques for Customizing Your STF” on page 37-44

“Create a Custom Target Configuration” on page 37-48

Control Code Generation With the System Target File

The system target file (STF) exerts overall control of the code generation stage of the build process. The STF also lets you control the presentation of your target to the end user. The STF provides

- Definitions of variables that are fundamental to the build process, such as the value for the `CodeFormat` TLC variable
- The main entry point to the top-level TLC program that generates code
- Target information for display in the System Target File Browser
- A mechanism for defining target-specific code generation options (and other parameters related to the build process) and for displaying them in the Configuration Parameters dialog box
- A mechanism for inheriting options from another target (such as the Embedded Real-Time (ERT) target)

Note that, although the STF is a Target Language Compiler (TLC) file, it contains embedded MATLAB code. Before creating or modifying an STF, you should acquire a working knowledge of TLC and of the MATLAB language. “Target Language Compiler” and “Scripts vs. Functions” describe the features and syntax of both the TLC and MATLAB languages.

While reading this section, you may want to refer to the STFs provided with the code generator. Most of these files are stored in the target-specific folders under *matlabroot* /

rtw/c (open). Additional STF files are stored under *matlabroot/toolbox/rtw/targets* (open).

System Target File Naming and Location Conventions

An STF must be located in a folder on the MATLAB path for the target to be displayed in the System Target File Browser and invoked in the build process. Follow the location and naming conventions for STFs and related target files given in “Folder and File Naming Conventions” on page 37-11.

System Target File Structure

- “Overview” on page 37-30
- “Header Comments” on page 37-32
- “TLC Configuration Variables” on page 37-33
- “TLC Program Entry Point and Related %includes” on page 37-34
- “RTW_OPTIONS Section” on page 37-35
- “rtwgensettings Structure” on page 37-35
- “Additional Code Generation Options” on page 37-37
- “Model Reference Considerations” on page 37-37

Overview

This section is a guide to the structure and contents of an STF. The following listing shows the general structure of an STF. Note that this is not a complete code listing of an STF. The listing consists of excerpts from each of the sections that make up an STF.

```
%%-----
%% Header Comments Section
%%-----
%% SYSTLC: Example Real-Time Target
%%   TMF: my_target.tmf MAKE: make_rtw EXTMODE: ext_comm
%% Initial comments contain directives for STF Browser.
%% Documentation, date, copyright, and other info may follow.
    ...
%selectfile NULL_FILE
    ...
%%-----
%% TLC Configuration Variables Section
%%-----
%% Assign code format, language, target type.
%%
%assign CodeFormat = "Embedded-C"
```

```

%assign TargetType = "RT"
%assign Language = "C"
%%
%%-----
%% TLC Program Entry Point
%%-----
%% Call entry point function.
%include "codegenentry.tlc"
%%
%%-----
%% (OPTIONAL) Generate Files for Build Process
%%-----
%include "mytarget_genfiles.tlc"
%%-----
%% RTW_OPTIONS Section
%%-----
/%
BEGIN_RTW_OPTIONS
%% Define rtwoptions structure array. This array defines target-specific
%% code generation variables, and controls how they are displayed.
rtwoptions(1).prompt = 'example code generation options';
...
rtwoptions(6).prompt = 'Show eliminated blocks';
rtwoptions(6).type = 'Checkbox';
...
%------%
% Configure RTW code generation settings %
%------%
...
%-----
%% rtwgensettings Structure
%%-----
%% Define suffix text for naming build folder here.
rtwgensettings.BuildDirSuffix = '_mytarget_rtw'
%% Callback compatibility declaration
rtwgensettings.Version = '1';

%% (OPTIONAL) target inheritance declaration
rtwgensettings.DerivedFrom = 'ert.tlc';
%% (OPTIONAL) other rtwGenSettings fields...
...
END_RTW_OPTIONS
%/
%%-----
%% targetComponentClass - MATHWORKS INTERNAL USE ONLY
%% REMOVE NEXT SECTION FROM USER_DEFINED CUSTOM TARGETS
%%-----
/%
BEGIN_CONFIGSET_TARGET_COMPONENT
targetComponentClass = 'Simulink.ERTTargetCC';
END_CONFIGSET_TARGET_COMPONENT
%/

```

If you are creating a custom target based on an existing STF, you must remove the `targetComponentClass` section (bounded by the directives

BEGIN_CONFIGSET_TARGET_COMPONENT and END_CONFIGSET_TARGET_COMPONENT). This section is reserved for the use of targets developed internally by MathWorks.

Header Comments

These lines at the head of the file are formatted as TLC comments. They provide required information to the System Target File Browser and to the build process. Note that you must place the browser comments at the head of the file, before other comments or TLC statements.

The presence of the comments enables the code generator to detect STFs. When the System Target File Browser is opened, the code generator scans the MATLAB path for TLC files that have formatted header comments. The comments contain the following directives:

- **SYSTLC**: The descriptor that appears in the browser.
- **TMF**: Name of the template makefile (TMF) to use during build process. When the target is selected, this filename is displayed in the “Template makefile” field of the **Code Generation** pane of the Configuration Parameters dialog box.
- **MAKE**: `make` command to use during build process. When the target is selected, this command is displayed in the **Make command** field of the **Code Generation** pane of the Configuration Parameters dialog box.
- **EXTMODE**: Name of external mode interface file (if any) associated with your target. If your target does not support external mode, use `no_ext_comm`.

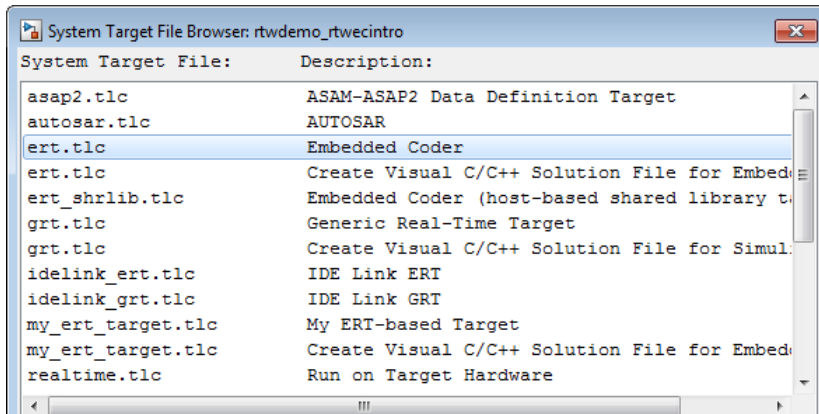
The following header comments are from `matlabroot/rtw/c/ert/ert.tlc`.

```
%% SYSTLC: Embedded Coder TMF: ert_default_tmf MAKE: make_rtw \  
%%   EXTMODE: ext_comm  
%% SYSTLC: Create Visual C/C++ Solution File for Embedded Coder\  
%%   TMF: RTW.MSVBuild MAKE: make_rtw EXTMODE: ext_comm  
.  
.  
.
```

Note: Limitation: Each comment can only contain a maximum of two lines, as shown in the preceding example.

Note that you can specify more than one group of directives in the header comments. Each such group is displayed as a different target configuration in the System Target File Browser. In the above example, the first two lines of code specify the default configuration of the ERT target. The next two lines specify a configuration that creates

and builds a Microsoft Visual C++ Solution (.sln) file. The figure below shows how these configurations appear in the System Target File Browser.



See “Create a Custom Target Configuration” on page 37-48 for an example of customized header comments.

TLC Configuration Variables

This section of the STF assigns global TLC variables that relate to the overall code generation process.

For an embedded target, in most cases you should simply use the global TLC variable settings used by the ERT target (`ert.tlc`). It is especially important that your STF use the 'Embedded-C' value for the `CodeFormat` TLC variable and uses the corresponding `rtwgensettings.DerivedFrom = 'ert.tlc'` in the `RTW_OPTIONS` section of the TLC file. Verify that values are assigned to the following variables:

- **CodeFormat:** The `CodeFormat` TLC variable selects generated code features. The 'Embedded-C' value for this variable is used by the ERT target. Your ERT-based target should specify 'Embedded-C' as the value for `CodeFormat`. This selection is designed for production code, minimal memory usage, static memory allocation, and a simplified interface to generated code.

For information on other values for the `CodeFormat` TLC variable, see “System Target Files and Code Generation Features” on page 17-18.

- **Language:** The only valid value is `C`, which enables support for C or C++ code generation as specified by the configuration parameter `TargetLang`.

- **TargetType**: The code generator defines the preprocessor symbols `RT` and `NRT` to distinguish simulation code from real-time code. These symbols are used in conditional compilation. The `TargetType` variable determines whether `RT` or `NRT` is defined.

Most targets are intended to generate real-time code. They assign `TargetType` as follows.

```
%assign TargetType = "RT"
```

Some targets, such as the model reference simulation target, accelerated simulation target, `RSim` target, and `S-function` target, generate code for use in nonreal time only. Such targets assign `TargetType` as follows.

```
%assign TargetType = "NRT"
```

TLC Program Entry Point and Related `%includes`

The code generation process normally begins with `codegenentry.tlc`. The STF invokes `codegenentry.tlc` as follows.

```
%include "codegenentry.tlc"
```

Note `codegenentry.tlc` and the lower-level TLC files assume that `CodeFormat`, `TargetType`, and `Language` have been assigned. Set these variables before including `codegenentry.tlc`.

If you need to implement target-specific code generation features, you should include the TLC file `mytarget_genfiles.tlc` in your STF. This file provides a mechanism for executing custom TLC code before and after invoking `codegenentry.tlc`. For information on this mechanism, see

- “Using `mytarget_genfiles.tlc`” on page 37-45 for an example of custom TLC code for execution after the main code generation entry point.
- “Target Development and the Build Process” on page 37-23 for general information on the build process, and for information on other build process customization hooks.

Another way to customize the code generation process is to call lower-level functions (normally invoked by `codegenentry.tlc`) directly, and include your own TLC functions at each stage of the process. This approach should be taken with caution. See “TLC Files” for more information.

The lower-level functions called by `codegenentry.tlc` are

- `genmap.tlc`: maps block names to corresponding language-specific block target files.
- `commonsetup.tlc`: sets up global variables.
- `commonentry.tlc`: starts the process of generating code.

RTW_OPTIONS Section

The `RTW_OPTIONS` section is bounded by the directives:

```
/%
  BEGIN_RTW_OPTIONS
  .
  .
  .
  END_RTW_OPTIONS
%/
```

The first part of the `RTW_OPTIONS` section defines an array of `rtwoptions` structures. This structure is discussed in “Using `rtwoptions` to Display Custom Target Options” on page 37-38.

The second part of the `RTW_OPTIONS` section defines `rtwgensettings`, a structure defining the build folder name and other settings for the code generation process. See “`rtwgensettings` Structure” on page 37-35 for information about `rtwgensettings`.

rtwgensettings Structure

The final part of the STF defines the `rtwgensettings` structure. This structure stores information that is written to the `model.rtw` file and used by the build process. The `rtwgensettings` fields of most interest to target developers are

- `rtwgensettings.Version`: Use this property to enable `rtwoptions` callbacks and to use the Callback API in `rtwgensettings.SelectCallback`.

Note: To use callbacks, you *must* set:

```
rtwgensettings.Version = '1';
```

Add the statement above to the **Configure RTW code generation settings** section of the system target file.

- `rtwgensettings.DerivedFrom`: This structure field defines the system target file from which options are to be inherited. See “Inheriting Target Options” on page 37-43.
- `rtwgensettings.SelectCallback`: This structure field specifies a `SelectCallback` function. You must set `rtwgensettings.Version = '1'`; or your callback will be ignored. `SelectCallback` is associated with the target rather than with any of its individual options. The `SelectCallback` function is triggered when the user selects a target with the System Target File browser.

The `SelectCallback` function is useful for setting up (or disabling) configuration parameters specific to the target.

The following code installs a `SelectCallback` function:

```
rtwgensettings.SelectCallback = 'my_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions.

Note If you have developed a custom target and you want it to be compatible with model referencing, you must implement a `SelectCallback` function to declare model reference compatibility. See “Support Model Referencing” on page 37-83.

- `rtwgensettings.ActivateCallback`: this property specifies an `ActivateCallback` function. The `ActivateCallback` function is triggered when the active configuration set of the model changes. This could happen during model loading, and also when the user changes the active configuration set.

The following code installs an `ActivateCallback` function:

```
rtwgensettings.ActivateCallback = 'my_activate_callback_handler(hDlg,hSrc)';
```

The arguments to the `ActivateCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions.

- `rtwgensettings.PostApplyCallback`: this property specifies a `PostApplyCallback` function. The `PostApplyCallback` function is triggered when the user clicks the **Apply** or **OK** button after editing options in the Configuration Parameters dialog box. The `PostApplyCallback` function is called after the changes have been applied to the configuration set.

The following code installs an `PostApplyCallback` function:

```
rtwgensettings.PostApplyCallback = 'my_postapply_callback_handler(hDlg,hSrc)';
```

The arguments to the `PostApplyCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions.

- `rtwgensettings.BuildDirSuffix`: Most targets define a folder name suffix that identifies build folders created by the target. The build process appends the suffix defined in the `rtwgensettings.BuildDirSuffix` field to the model name to form the name of the build folder. For example, if you define `rtwgensettings.BuildDirSuffix` as follows

```
rtwgensettings.BuildDirSuffix = '_mytarget_rtw'
```

the build folders are named `model_mytarget_rtw`.

Additional Code Generation Options

“Configure Generated Code with TLC” on page 35-22 describes additional TLC code generation variables. End users of a target can assign these variables by entering a MATLAB command of the form

```
set_param(modelName,'TLCOptions','-aVariable=val');
```

(For more information, see “Specify TLC Options” on page 25-22.)

However, the preferred approach is to assign these variables in the STF using statements of the form:

```
%assign Variable = val
```

For readability, we recommend that you add such assignments in the section of the STF after the comment **Configure RTW code generation settings**.

Model Reference Considerations

See “Support Model Referencing” on page 37-83 for important information on STF and other modifications you may need to make to support the code generator model referencing features.

Define and Display Custom Target Options

- “Using rtwoptions to Display Custom Target Options” on page 37-38
- “Example System Target File With Customized rtwoptions” on page 37-42

- “Inheriting Target Options” on page 37-43

Using `rtwoptions` to Display Custom Target Options

You control the options to display in the **Code Generation** pane of the Configuration Parameters dialog box by customizing the `rtwoptions` structure in your system target file.

The fields of the `rtwoptions` structure define variables and associated user interface elements to be displayed in the Configuration Parameters dialog box. Using the `rtwoptions` structure array, you can define target-specific options displayed in the dialog box and organize options into categories. You can also write callback functions to specify how these options are processed.

When the **Code Generation** pane opens, the `rtwoptions` structure array is scanned and the listed options are displayed. Each option is represented by an assigned user interface element (check box, edit field, menu, or push button), which displays the current option value.

The user interface elements can be in an enabled or disabled (grayed-out) state. If an option is enabled, the user can change the option value.

You can also use the `rtwoptions` structure array to define special NonUI elements that cause callback functions to be executed, but that are not displayed in the **Code Generation** pane. See “NonUI Elements” on page 37-42 for details.

The elements of the `rtwoptions` structure array are organized into groups. Each group of items begins with a header element of type **Category**. The default field of a **Category** header must contain a count of the remaining elements in the category.

The **Category** header is followed by options to be displayed on the **Code Generation** pane. The header in each category is followed by one or more option definition elements.

Each category of target options corresponds to options listed under **Code Generation** in the Configuration Parameters dialog box.

The table `rtwoptions` Structure Fields Summary summarizes the fields of the `rtwoptions` structure.

Example `rtwoptions` Structure

The following `rtwoptions` structure is excerpted from an example system target file, `matlabroot/toolbox/rtw/rtwdemos/rtwoptions_demo/usertarget.tlc`. The

code defines an `rtwoptions` structure array. The default field of the first (header) element is set to 4, indicating the number of elements that follow the header.

```

rtwoptions(1).prompt      = 'userPreferred target options (I)';
rtwoptions(1).type        = 'Category';
rtwoptions(1).enable      = 'on';
rtwoptions(1).default     = 4; % number of items under this category
                           % excluding this one.
rtwoptions(1).popupstrings = ''; % At the first item, user has to
rtwoptions(1).tlcvariable = ''; % initialize all supported fields
rtwoptions(1).tooltip     = '';
rtwoptions(1).callback    = '';
rtwoptions(1).makevariable = '';

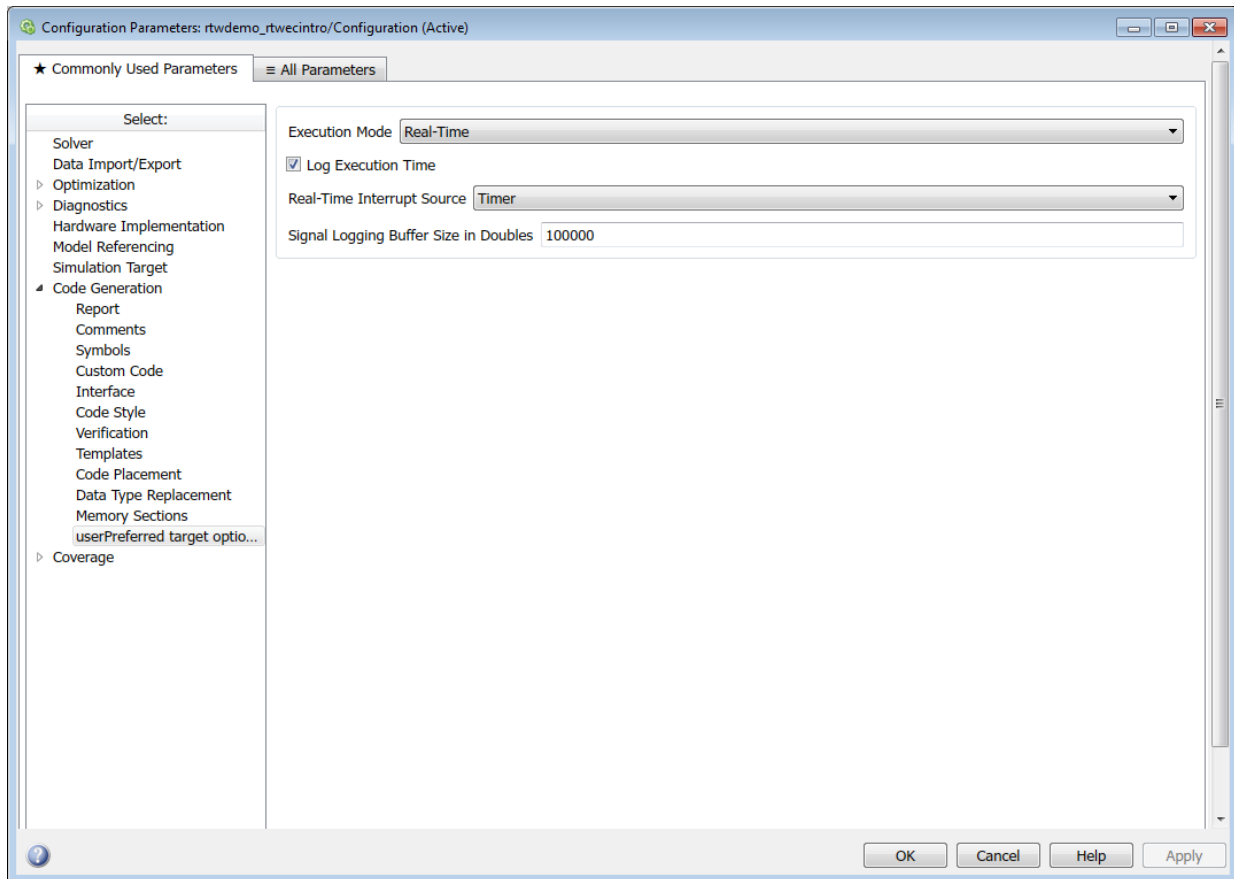
rtwoptions(2).prompt      = 'Execution Mode';
rtwoptions(2).type        = 'Popup';
rtwoptions(2).default     = 'Real-Time';
rtwoptions(2).popupstrings = 'Real-Time|UserDefined';
rtwoptions(2).tlcvariable = 'tlcvariable1';
rtwoptions(2).tooltip     = ['See this text as tooltip'];

rtwoptions(3).prompt      = 'Log Execution Time';
rtwoptions(3).type        = 'Checkbox';
rtwoptions(3).default     = 'on';
rtwoptions(3).tlcvariable = 'RL32LogTETModifier';
rtwoptions(3).tooltip     = ['']; % no tooltip

rtwoptions(4).prompt      = 'Real-Time Interrupt Source';
rtwoptions(4).type        = 'Popup';
rtwoptions(4).default     = 'Timer';
rtwoptions(4).popupstrings = 'Timer|5|6|7|8|9|10|11|12|13|14|15';
rtwoptions(4).tlcvariable = 'tlcvariable3';
rtwoptions(4).callback    = 'usertargetcallback(hDlg, hSrc, ''tlcvariable3'')';
rtwoptions(4).tooltip     = [''];
rtwoptions(4).tooltip     = ['See TLC file for how to use reserved '...
    ' keyword ''hDlg'', and ''hSrc''.'];
...
rtwoptions(5).prompt      = 'Signal Logging Buffer Size in Doubles';
rtwoptions(5).type        = 'Edit';
rtwoptions(5).default     = '100000';
rtwoptions(5).tlcvariable = 'tlcvariable2';
rtwoptions(5).tooltip     = [''];

```

The first element adds a **userPreferred target options (I)** pane under **Code Generation** in the Configuration Parameters dialog box. The pane displays the options defined in `rtwoptions(2)`, `rtwoptions(3)`, `rtwoptions(4)`, and `rtwoptions(5)`.



If you want to define a large number of options, you can define multiple **Category** groups within a single system target file.

Note the `rtwoptions` structure and callbacks are written in MATLAB code, although they are embedded in a TLC file. To verify the syntax of your `rtwoptions` structure definitions and code, you can execute the commands at the MATLAB prompt by copying and pasting them to the MATLAB Command Window.

To learn more about `usertarget.tlc` and the example callback files provided with it, see “Example System Target File With Customized `rtwoptions`” on page 37-42. For more examples of target-specific `rtwoptions` definitions, see the `target.tlc` files under `matlabroot/rtw/c` (open).

rtwoptions Structure Fields Summary lists the fields of the `rtwoptions` structure.

rtwoptions Structure Fields Summary

Field Name	Description
<code>callback</code>	For examples of callback usage, see “Example System Target File With Customized <code>rtwoptions</code> ” on page 37-42.
<code>closecallback</code> (obsolete)	Do not use <code>closecallback</code> . Use <code>rtwgensettings.PostApplyCallback</code> instead (see “ <code>rtwgensettings</code> Structure” on page 37-35). <code>closecallback</code> is ignored. For examples of callback usage, see “Example System Target File With Customized <code>rtwoptions</code> ” on page 37-42.
<code>default</code>	Default value of the option (empty if the <code>type</code> is <code>Pushbutton</code>).
<code>enable</code>	Must be 'on' or 'off'. If 'on', the option is displayed as an enabled item; otherwise, as a disabled item.
<code>makevariable</code>	Template makefile token (if any) associated with the option. The <code>makevariable</code> is expanded during processing of the template makefile. See “Template Makefile Tokens” on page 37-62.
<code>modelReferenceParameter-Check</code>	Specifies whether the option must have the same value in a referenced model and its parent model. If this field is unspecified or has the value 'on' the option values must be same. If the field is specified and has the value 'off' the option values can differ. See “Controlling Configuration Option Value Agreement” on page 37-88.
<code>NonUI</code>	Element that is not displayed, but is used to invoke a close or open callback. See “NonUI Elements” on page 37-42.
<code>opencallback</code> (obsolete)	Do not use <code>opencallback</code> . Use <code>rtwgensettings.SelectCallback</code> instead (see “ <code>rtwgensettings</code> Structure” on page 37-35). For examples of callback usage, see “Example System Target File With Customized <code>rtwoptions</code> ” on page 37-42.
<code>popupstrings</code>	If <code>type</code> is <code>Popup</code> , <code>popupstrings</code> defines the items in the menu. Items are delimited by the " " (vertical bar) character. The

Field Name	Description
	following example defines the items of the MAT-file variable name modifier menu used by the GRT target. 'rt_ _rt none'
prompt	Label for the option.
tlcvariable	Name of TLC variable associated with the option.
tooltip	Help text displayed when mouse is over the item.
type	Type of element: Checkbox, Edit, NonUI, Popup, Pushbutton, or Category.

NonUI Elements

Elements of the `rtwoptions` array that have type `NonUI` exist solely to invoke callbacks. A `NonUI` element is not displayed in the Configuration Parameters dialog box. You can use a `NonUI` element if you want to execute a callback that is not associated with a user interface element, when the dialog box opens or closes. See the next section, “Example System Target File With Customized `rtwoptions`” on page 37-42 for an example.

Note: The default value of a `NonUI` element determines the set of values allowed for that element.

- If the default value is '0' or '1', the element stores a Boolean value.
 - If the default value contains an integer other than '0' or '1', the element stores a value of type `int32`.
 - If the default value does not contain an integer, the element is evaluated as a character vector.
-

Example System Target File With Customized `rtwoptions`

A working system target file, with MATLAB file callback functions, has been provided as an example of how to use the `rtwoptions` structure to display and process custom options on the **Code Generation** pane. The examples are compatible with the callback API.

The example target files are in the folder (open):

`matlabroot/toolbox/rtw/rtwdemos/rtwoptions_demo`

The example target files include:

- `usertarget.tlc`: The example system target file. This file illustrates how to define custom menus, check boxes, and edit fields. The file also illustrates the use of callbacks.
- `usertargetcallback.m`: A MATLAB file callback invoked by a menu.

Refer to the example files while reading this section. The example system target file, `usertarget.tlc`, illustrates the use of `rtwoptions` to display the following custom target options:

- The **Execution Mode** menu.
- The **Log Execution Time** check box.
- The **Real-Time Interrupt Source** menu. The menu executes a callback defined in an external file, `usertargetcallback.m`. The TLC variable associated with the menu is passed in to the callback, which displays the menu's current value.
- The edit field **Signal Logging Buffer Size in Doubles**.

Try studying the example code while interacting with the example target options in the Configuration Parameters dialog box. To interact with the example target file,

- 1 Make `matlabroot/toolbox/rtw/rtwdemos/rtwoptions_demo` (open) your working folder.
- 2 Open a model of your choice.
- 3 Open the Configuration Parameters dialog box and select the **Code Generation** pane.
- 4 Click **Browse**. The System Target File Browser opens. Select `usertarget.tlc`. Then click **OK**.
- 5 Observe that the **Code Generation** pane contains a custom sub-tab: **userPreferred target options (I)**.
- 6 As you interact with the options in this category and open and close the Configuration Parameters dialog box, observe the messages displayed in the MATLAB Command Window. These messages are printed from code in the STF, or from callbacks invoked from the STF.

Inheriting Target Options

`ert.tlc` provides a basic set of Embedded Coder code generation options. If your target is based on `ert.tlc`, your STF should normally inherit the options defined in ERT.

Use the `rtwgensettings.DerivedFrom` field in the `rtwgensettings` structure to define the system target file from which options are to be inherited. You should convert your custom target to use this mechanism as follows.

Set the `rtwgensettings.DerivedFrom` field value as in the following example:

```
rtwgensettings.DerivedFrom = 'stf.tlc';
```

where `stf` is the name of the system target file from which options are to be inherited. For example:

```
rtwgensettings.DerivedFrom = 'ert.tlc';
```

When the Configuration Parameters dialog box executes this line of code, it includes the options from `stf.tlc` automatically. If `stf.tlc` is a MathWorks internal system target file that has been converted to a new layout, the dialog box displays the inherited options using the new layout.

Handling Unsupported Options

If your target does not support all of the options inherited from `ert.tlc`, you should detect unsupported option settings and display a warning or error message. In some cases, if a user has selected an option your target does not support, you may need to abort the build process. For example, if your target does not support the **Generate an example main program** option, the build process should not be allowed to proceed if that option is selected.

Even though your target may not support all inherited ERT options, it is required that the ERT options are retained in the **Code Generation** pane of the Configuration Parameters dialog box. Do not simply remove unsupported options from the `rtwoptions` structure in the STF. Options must be in the dialog box to be scanned by the code generator when it performs optimizations.

For example, you may want to prevent users from turning off the **Single output/update function** option. It may seem reasonable to remove this option from the dialog box and simply assign the TLC variable `CombineOutputUpdateFcns` to `on`. However, if the option is not included in the dialog box, the code generator assumes that output and update functions are *not* to be combined. Less efficient code is generated as a result.

Tips and Techniques for Customizing Your STF

- “Introduction” on page 37-45
- “Required and Recommended %includes” on page 37-45

- “Handling Aliases for Target Option Values” on page 37-46
- “Supporting Multiple Development Environments” on page 37-48

Introduction

The following sections include information on techniques for customizing your STF, including

- How to invoke custom TLC code from your STF
- Approaches to supporting multiple development environments

Required and Recommended %includes

If you need to implement target-specific code generation features, we recommend that your STF include the TLC file `mytarget_genfiles.tlc`.

Once your STF has set up the required TLC environment, you must include `codegenentry.tlc` to start the standard code generation process.

`mytarget_genfiles.tlc` provides a mechanism for executing custom TLC code after the main code generation entry point. See “Using `mytarget_genfiles.tlc`” on page 37-45.

Using `mytarget_genfiles.tlc`

`mytarget_genfiles.tlc` (optional) is useful as a central file from which to invoke target-specific TLC files that generate additional files as part of your target build process. For example, your target may create sub-makefiles or project files for a development environment, or command scripts for a debugger to do automatic downloads.

The build process can then invoke these generated files either directly from the make process, or after the executable is created. This is done with the `STF_make_rtw_hook.m` mechanism, as described in “Customize Build Process with `STF_make_rtw_hook` File” on page 35-29.

The following TLC code shows an example `mytarget_genfiles.tlc` file.

```
%selectfile NULL_FILE

%assign ModelName = CompiledModel.Name

%% Create Debugger script
```

```
%assign model_script_file = "%<ModelName>.cfg"  
%assign script_file = "debugger_script_template.tlc"  
  
%if RTWVerbose  
    %selectfile STDOUT  
    ### Creating %<model_script_file>  
    %selectfile NULL_FILE  
%endif  
  
%include "%<script_file>"  
%openfile bld_file = "%<model_script_file>"  
%<CreateDebuggerScript()>  
%closefile bld_file
```

Handling Aliases for Target Option Values

This section describes utility functions that can be used to detect and resolve alias values or legacy values when testing user-specified values for the target device type (`ProdHWDeviceType`) and the code replacement library (`CodeReplacementLibrary`).

RTW.isHWDeviceTypeEq

To test if two target device type values represent the same hardware device, invoke the following function:

```
result = RTW.isHWDeviceTypeEq(type1, type2)
```

where *type1* and *type2* are character vectors containing target device type values or aliases.

The `RTW.isHWDeviceTypeEq` function returns true if *type1* and *type2* are character vectors representing the same hardware device. For example, the following call returns true:

```
RTW.isHWDeviceTypeEq('Specified', 'Generic->Custom')
```

For a description of the target device type option `ProdHWDeviceType`, see the command-line information for the **Hardware Implementation** pane parameters “Device vendor” and “Device type” in the Simulink reference documentation.

RTW.resolveHWDeviceType

To return the device type value for a hardware device, given a value that might be an alias or legacy value, invoke the following function:

```
result = RTW.resolveHWDeviceType(type)
```

where *type* is a character vector containing a target device type value or alias.

The `RTW.resolveHWDeviceType` function returns the device type value of the device. For example, the following calls both return `'Generic->Custom'`:

```
RTW.resolveHWDeviceType('Specified')
RTW.resolveHWDeviceType('Generic->Custom')
```

For a description of the target device type option `ProdHWDeviceType`, see the command-line information for the **Hardware Implementation** pane parameters “Device vendor” and “Device type” in the Simulink reference documentation.

RTW.isTf1Eq

To test if two code replacement library (CRL) names represent the same CRL, invoke the following function:

```
result = RTW.isTf1Eq(name1,name2)
```

where *name1* and *name2* are character vectors containing CRL values or aliases.

The `RTW.isTf1Eq` function returns true if *name1* and *name2* are character vectors representing the same code replacement library. For example, the following call returns true:

```
RTW.isTf1Eq('GNU','GNU C99 extensions')
```

For a description of the `CodeReplacementLibrary` parameter, see “Code replacement library”.

RTW.resolveTf1Name

To return the CRL value for a code replacement library, given a value that might be an alias or legacy value, invoke the following function:

```
result = RTW.resolveTf1Name(name)
```

where *name* is a character vector containing a CRL value or alias.

The `RTW.resolveTf1Name` function returns the value of the referenced code replacement library. For example, the following calls both return `'GNU C99 extensions'`:

```
RTW.resolveTf1Name('GNU')
RTW.resolveTf1Name('GNU C99 extensions')
```

For a description of the `CodeReplacementLibrary` parameter, see “Code replacement library”.

Supporting Multiple Development Environments

Your target may require support for multiple development environments (for example, two or more cross-compilers) or multiple modes of code generation (for example, generating a binary executable versus generating a project file for your compiler).

One approach to this requirement is to implement multiple STF's. Each STF invokes a template makefile for the development environment. This amounts to providing two separate targets.

Create a Custom Target Configuration

- “Introduction” on page 37-48
- “my_ert_target Overview” on page 37-49
- “Creating Target Folders” on page 37-50
- “Create ERT-Based, Toolchain Compliant STF” on page 37-51
- “Create ERT-Based TMF” on page 37-58
- “Create Test Model and S-Function” on page 37-58
- “Verify Target Operation” on page 37-60

Introduction

This tutorial can supplement the example target guides described in “Sample Custom Targets” on page 37-9. For an introduction and example files, try the example targets first.

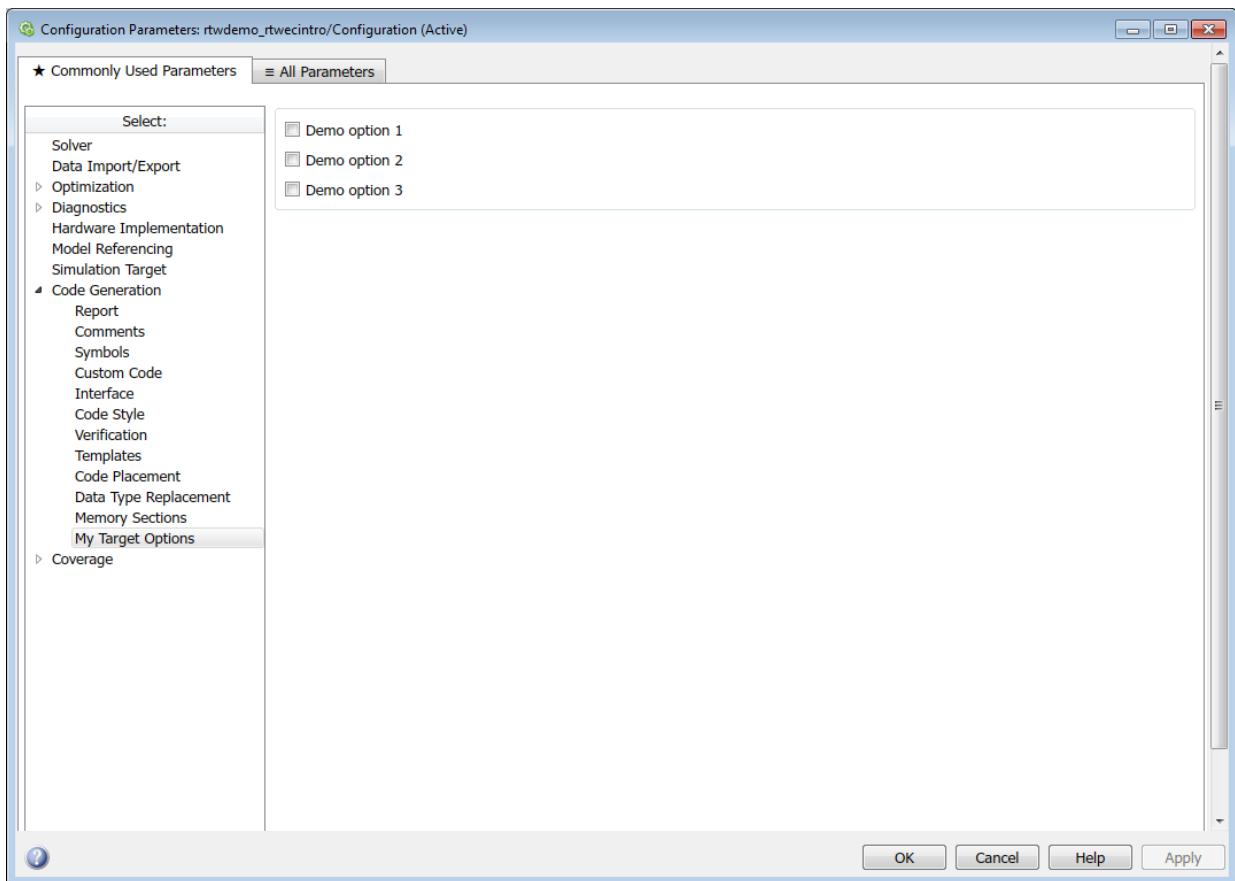
This tutorial guided you through the process of creating an ERT-based target, `my_ert_target`. This exercise illustrates several tasks, which are typical for creating a custom target:

- Setting up target folders and modifying the MATLAB path.
- Making modifications to a standard STF and TMF such that the custom target is visible in the System Target File Browser, inherits ERT options, displays target-specific options, and generates code with the default host-based compiler.
- Testing the build process with the custom target, using a simple model that incorporates an inlined S-function.

During this exercise, you implement an operational, but skeletal, ERT-based target. This target can be useful as a starting point in a complete implementation of a custom embedded target.

my_ert_target Overview

In the following sections, you create a skeletal target, `my_ert_target`. The target inherits and supports the standard options of the ERT target and displays additional target-specific options in the Configuration Parameters dialog box (see Target-Specific Options for `my_ert_target`).



Target-Specific Options for my_ert_target

`my_ert_target` supports a toolchain-based build, generating code and executables that run on the host system. `my_ert_target` uses the `lcc` compiler on a Microsoft Windows platform. The chosen compiler is readily available and is distributed with the code generator. On a Microsoft Windows platform, if you use a different compiler, you can set up `lcc` temporarily as your default compiler through the following MATLAB command:

```
mex -setup
```

The software displays links for supported compilers that are installed on your computer. Click the `lcc` link.

Note On Linux systems, make sure that you have an installed C compiler. If so, you can use Linux folder syntax to complete this exercise.

`my_ert_target` can also support template makefile-based builds. For more information about using this target with the template makefile approach, see “Create ERT-Based TMF” on page 37-58.

You can test `my_ert_target` with a model that is compatible with the ERT target (see “Select an ERT System Target File”). Generated programs operate identically to ERT generated programs.

To simplify the testing of your target, test with `targetmodel`, a very simple fixed-step model (see “Create Test Model and S-Function” on page 37-58). The S-Function block in `targetmodel` uses the source code from the `timestwo` example, and generates fully inlined code. See “S-Function Examples” and “Inline S-Functions with TLC” for further discussion of the `timestwo` example S-function.

Creating Target Folders

Create folders to store the target files and add them to the MATLAB path, following the recommended conventions (see “Folder and File Naming Conventions” on page 37-11). You also create a folder to store the test model, S-function, and generated code.

This example assumes that your target and model folders are located within the folder `c:/work`. Do not place your target and model folders within the MATLAB folder tree (that is, in or under the `matlabroot` folder).

To create the folders and make them accessible:

- 1 Create a target root folder, `my_ert_target`. From the MATLAB Command Window on a Windows platform, enter:

```
cd c:/work
mkdir my_ert_target
```

- 2 Within the target root folder, create a subfolder to store your target files.

```
mkdir my_ert_target/my_ert_target
```

- 3 Add these folders to your MATLAB path.

```
addpath c:/work/my_ert_target
addpath c:/work/my_ert_target/my_ert_target
```

- 4 Create a folder, `my_targetmodel`, to store the test model, S-function, and generated code.

```
mkdir my_targetmodel
```

Create ERT-Based, Toolchain Compliant STF

Create an STF for your target by copying and modifying the standard STF for the ERT target. Then, validate the STF by viewing the new target in the System Target File Browser and in the Configuration Parameters dialog box.

Editing the STF

To edit the STF, use these steps:

- 1 Change your working folder to the folder you created in “Creating Target Folders” on page 37-50.

```
cd c:/work/my_ert_target/my_ert_target
```

- 2 Place a copy of `matlabroot/rtw/c/ert/ert.tlc` in `c:/work/my_ert_target/my_ert_target` and rename it to `my_ert_target.tlc`. The file `ert.tlc` is the STF for the ERT target.

- 3 Open `my_ert_target.tlc` in a text editor of your choice.

- 4 Customize the STF, replacing the header comment lines with directives that make your STF visible in the System Target File Browser and define the associated TMF, `make` command, and external mode interface file (if any). For more information about these directives, see “Header Comments” on page 37-32 .

Replace the header comments in `my_ert_target.tlc` with the following header comments.

```
%% SYSTLC: My ERT-based Target TMF: my_ert_target_lcc.tmf MAKE: make_rtw \
```

```
%% EXTMODE: no_ext_comm
```

- 5 The file `my_ert_target.tlc` inherits the standard ERT options, using the mechanism described in “Inheriting Target Options” on page 37-43. Therefore, the existing `rtwoptions` structure definition is superfluous. Edit the `RTW_OPTIONS` section such that it includes only the following code.

```
/%
BEGIN_RTW_OPTIONS

%-----%
% Configure RTW code generation settings %
%-----%

rtwgenSettings.BuildDirSuffix = '_ert_rtw';

END_RTW_OPTIONS
%/
```

- 6 Delete the code after the end of the `RTW_OPTIONS` section, which is delimited by the directives `BEGIN_CONFIGSET_TARGET_COMPONENT` and `END_CONFIGSET_TARGET_COMPONENT`. This code is for use only by internal MathWorks developers.
- 7 Modify the build folder suffix in the `rtwgenSettings` structure in accordance with the conventions described in “`rtwgenSettings` Structure” on page 37-35.

To set the suffix to a character vector for the `_my_ert_target` custom target, change the line

```
rtwgenSettings.BuildDirSuffix = '_ert_rtw'
```

to

```
rtwgenSettings.BuildDirSuffix = '_my_ert_target_rtw'
```

- 8 Modify the `rtwgenSettings` structure to inherit options from the ERT target and declare Release 14 or later compatibility as described in “`rtwgenSettings` Structure” on page 37-35. Add the following code to the `rtwgenSettings` definition:

```
rtwgenSettings.DerivedFrom = 'ert.tlc';
rtwgenSettings.Version = '1';
```

- 9 Add an `rtwoptions` structure that defines a target-specific options category with three check boxes just after the `BEGIN_RTW_OPTIONS` directive. The following code shows the complete `RTW_OPTIONS` section, including the previous `rtwgenSettings` changes.

```
/%
BEGIN_RTW_OPTIONS
```

```

rtwoptions(1).prompt      = 'My Target Options';
rtwoptions(1).type        = 'Category';
rtwoptions(1).enable      = 'on';
rtwoptions(1).default     = 3; % number of items under this category
                           % excluding this one.
rtwoptions(1).popupstrings = '';
rtwoptions(1).tlcvariable = '';
rtwoptions(1).tooltip     = '';
rtwoptions(1).callback    = '';
rtwoptions(1).makevariable = '';

rtwoptions(2).prompt      = 'Demo option 1';
rtwoptions(2).type        = 'Checkbox';
rtwoptions(2).default     = 'off';
rtwoptions(2).tlcvariable = 'DummyOpt1';
rtwoptions(2).makevariable = '';
rtwoptions(2).tooltip     = ['Demo option1 (non-functional)'];
rtwoptions(2).callback    = '';

rtwoptions(3).prompt      = 'Demo option 2';
rtwoptions(3).type        = 'Checkbox';
rtwoptions(3).default     = 'off';
rtwoptions(3).tlcvariable = 'DummyOpt2';
rtwoptions(3).makevariable = '';
rtwoptions(3).tooltip     = ['Demo option2 (non-functional)'];
rtwoptions(3).callback    = '';

rtwoptions(4).prompt      = 'Demo option 3';
rtwoptions(4).type        = 'Checkbox';
rtwoptions(4).default     = 'off';
rtwoptions(4).tlcvariable = 'DummyOpt3';
rtwoptions(4).makevariable = '';
rtwoptions(4).tooltip     = ['Demo option3 (non-functional)'];
rtwoptions(4).callback    = '';

%-----%
% Configure RTW code generation settings %
%-----%

rtwgensettings.BuildDirSuffix = '_my_ert_target_rtw';
rtwgensettings.DerivedFrom = 'ert.tlc';
rtwgensettings.Version = '1';
rtwgensettings.SelectCallback = 'enableToolchainCompliant(hSrc, hDlg)';
%SelectCallback provides toolchain approach support, but requires custom function
%Omit this SelectCallback if using the template makefile approach

END_RTW_OPTIONS
%/

```

10 Save your changes to `my_ert_target.tlc` and close the file.

Create ToolchainCompliant Function

To enable builds using the toolchain approach, you create a function that corresponds to the `SelectCallback` near the end of the custom STF. This function sets properties for toolchain compliance.

```
function enableToolchainCompliant(hSrc, hDlg)
    hCS = hSrc.getConfigSet();

    % The following parameters enable toolchain compliance.
    slConfigUISetVal(hDlg, hSrc, 'UseToolchainInfoCompliant', 'on');
    hCS.setProp('GenerateMakefile','on');

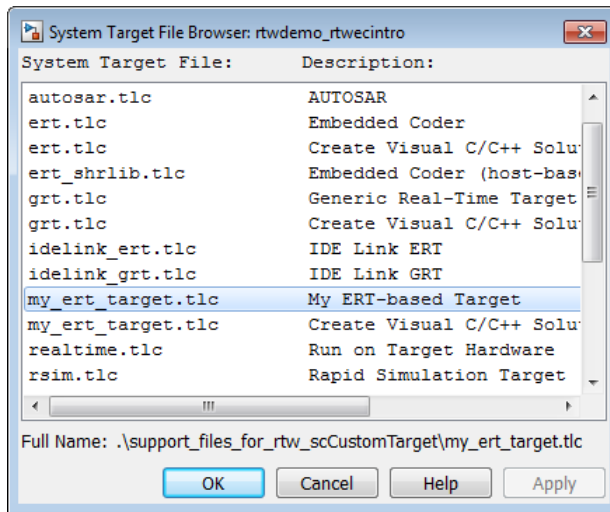
    % The following parameters are not required for toolchain compliance.
    % But, it is recommended practice to set these default values and
    % disable the parameters (as shown).
    hCS.setProp('RTWCompilerOptimization','off');
    hCS.setProp('MakeCommand','make_rtw');
    hCS.setPropEnabled('RTWCompilerOptimization',false);
    hCS.setPropEnabled('MakeCommand',false);
end
```

Note: If you are using the template makefile approach, omit calling the function enabling toolchain-compliance from your STF file. Instead, use the information in “Create ERT-Based TMF” on page 37-58.

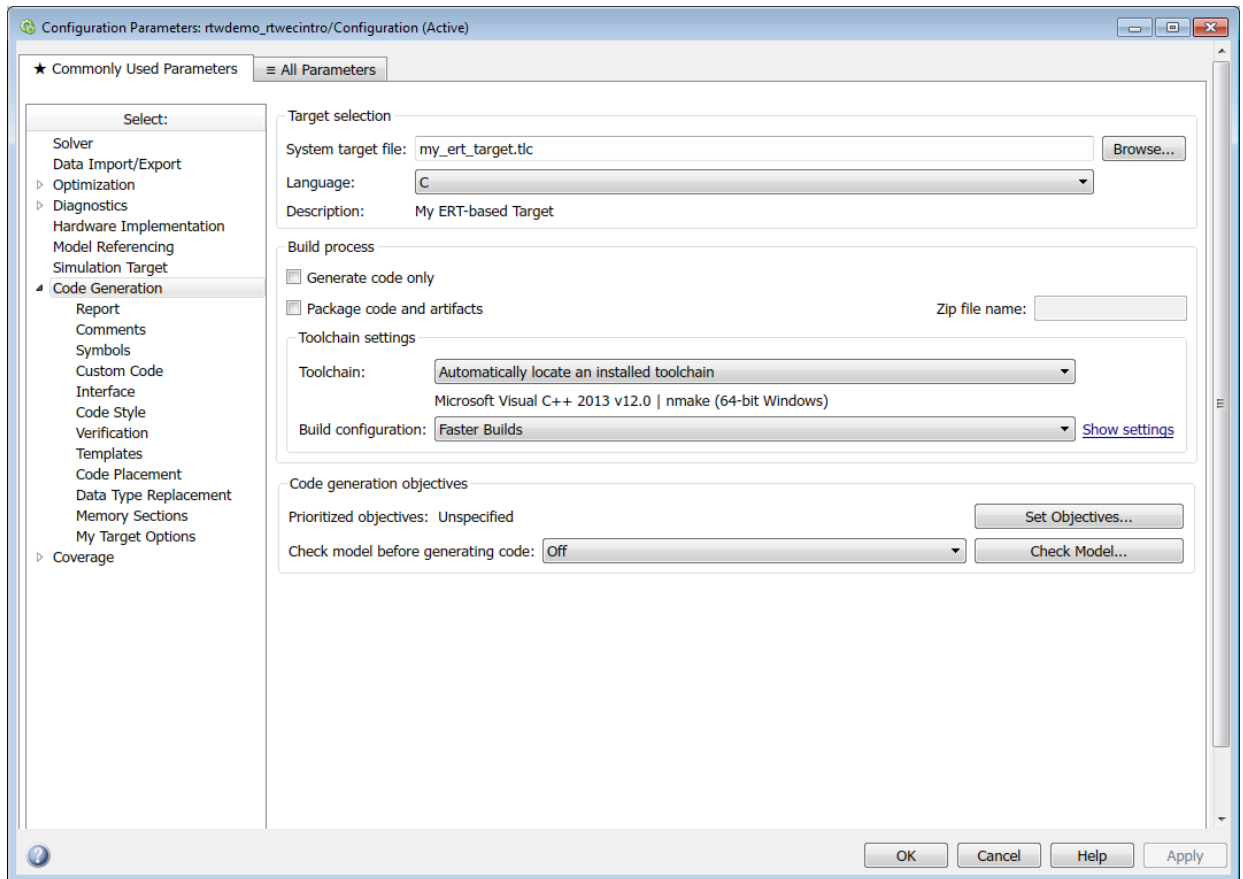
Viewing the STF

At this point, you can verify that the target inherits and displays ERT options as follows:

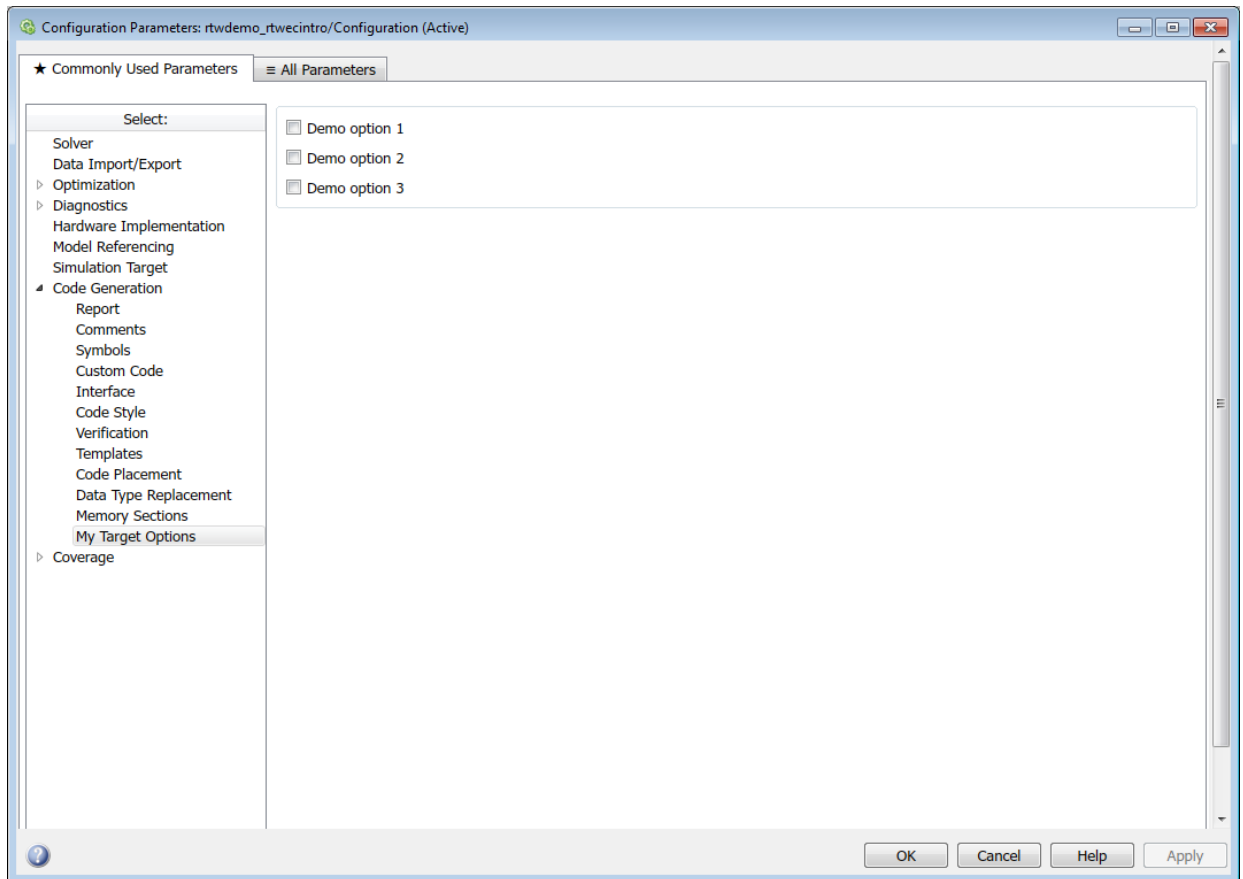
- 1 Create a new model.
- 2 Open the Model Explorer or the Configuration Parameters dialog box.
- 3 Select the **Code Generation** pane.
- 4 Click **Browse** to open the System Target File browser.
- 5 In the file browser, scroll through the list of targets to find the new target, `my_ert_target.tlc`. (This step assumes that your MATLAB path contains `c:/work/my_ert_target/my_ert_target`, as previously set in “Creating Target Folders” on page 37-50.)
- 6 Select My ERT-based Target and click **OK**.



- 7 The **Code Generation** pane now shows that the model is configured for the `my_ert_target.tlc` target. The **System target file**, **Language**, **Toolchain**, and **Build configuration** fields should appear:



- 8 Select the **My Target Options** pane. The target displays the three check box options defined in the `rtwoptions` structure.



- 9 Select the **Code Generation** pane and reopen the System Target File Browser.
- 10 Select the Embedded Coder target (`ert.tlc`). The target displays the standard ERT options.
- 11 Close the model. You do not need to save it.

The STF for the skeletal target is complete. If you are using the toolchain approach, you are ready to invoke the build process for your target.

If you prefer to use the template makefile approach, the reference to a TMF, `my_ert_target_lcc.tmf`, in the STF header comments prevents you from invoking

the build process for your target until the TMF file is in place. First, you must create a `my_ert_target_lcc.tmf` file.

Create ERT-Based TMF

If you are using the toolchain makefile approach with a toolchain compliant custom target, omit the steps that apply to the template makefile approach. (Skip this section.)

If you are using the templated makefile approach, follow the steps applying to TMF and omit calling the function enabling toolchain-compliance from your STF file, which is described in “Create ERT-Based, Toolchain Compliant STF” on page 37-51.

Create a TMF for your target by copying and modifying the standard ERT TMF for the LCC compiler:

- 1 Make sure that your working folder is still set to the target file folder you created previously in “Creating Target Folders” on page 37-50.

```
c:/work/my_ert_target/my_ert_target
```

- 2 Place a copy of `matlabroot/rtw/c/ert/ert_lcc.tmf` in `c:/work/my_ert_target/my_ert_target` and rename it `my_ert_target_lcc.tmf`. The file `ert_lcc.tmf` is the ERT compiler-specific template makefile for the LCC compiler.

- 3 Open `my_ert_target_lcc.tmf` in a text editor.

- 4 Change the `SYS_TARGET_FILE` parameter so that the file reference for your `.tlc` file is generated in the make file. Change the line

```
SYS_TARGET_FILE = any
```

to

```
SYS_TARGET_FILE = my_ert_target.tlc
```

- 5 Save changes to `my_ert_target_lcc.tmf` and close the file.

Your target can now generate code and build a host-based executable. In the next sections, you create a test model and test the build process using `my_ert_target`.

Create Test Model and S-Function

In this section, you build a simple test model for later use in code generation:

- 1 Set your working folder to `c:/work/my_targetmodel`.


```
cd c:/work/my_targetmodel
```

For the remainder of this tutorial, `my_targetmodel` is assumed to be the working folder. Your target writes the output files of the code generation process into a build folder within the working folder. When inlined code is generated for the `timestwo` S-function, the build process looks for the TLC implementation of the S-function in the working folder.

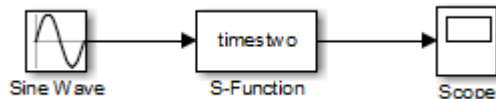
- 2 Copy the following C and TLC files for the `timestwo` S-function to your working folder:

- `matlabroot/toolbox/simulink/simdemos/simfeatures/src/timestwo.c`
- `matlabroot/toolbox/simulink/simdemos/simfeatures/tlc_c/timestwo.tlc`

- 3 Build the `timestwo` MEX-file in `c:/work/my_targetmodel`.

```
mex timestwo.c
```

- 4 Create the following model, using an S-Function block from the Simulink User-Defined Functions library. Save the model in your working folder as `targetmodel`.



- 5 Double-click the S-Function block to open the Block Parameters dialog box. Enter the S-function name `timestwo`. Click **OK**. The block is now bound to the `timestwo` MEX-file.
- 6 Open Model Explorer or the Configuration Parameters dialog box and select the **Solver** pane.
- 7 Set the solver **Type** to `fixed-step` and click **Apply**.
- 8 Save the model.
- 9 Open the scope and run a simulation. Verify that the `timestwo` S-function multiplies its input by 2.0.

Keep the `targetmodel` model open for use in the next section, in which you generate code using the test model.

Verify Target Operation

In this section you configure `targetmodel` for the `my_ert_target` custom target, and use the target to generate code and build an executable:

- 1 Open the Configuration Parameters dialog box and select the **Code Generation** pane.
- 2 Click **Browse** to open the System Target File Browser.
- 3 In the Browser, select **My ERT-based Target** and click **OK**.
- 4 The Configuration Parameters dialog box now displays the **Code Generation** pane for `my_ert_target`.
- 5 Select the **Code Generation > Report** pane and select the **Create code generation report** option.
- 6 Click **Apply** and save the model. The model is configured for `my_ert_target`.
- 7 Build the model. If the build succeeds, the MATLAB Command Window displays the message below.

```
### Created executable: ../targetmodel.exe
### Successful completion of build procedure for model:
targetmodel
```

Your working folder contains the `targetmodel.exe` file and the build folder, `targetmodel_my_ert_target_rtw`, which contains generated code and other files. The working folder also contains an `slprj` folder, used internally by the build process.

The code generator also creates and displays a code generation report.

- 8 To view the generated model code, go to the code generation report window. In the **Contents** pane, click the `targetmodel.c` link.
- 9 In `targetmodel.c`, locate the model step function, `targetmodel_step`. Observe the following code.

```
/* S-Function Block: <Root>/S-Function */
/* Multiply input by two */
targetmodel_B.SFunction = targetmodel_B.SineWave * 2.0;
```

The presence of this code confirms that the `my_ert_target` custom target has generated an inlined output computation for the S-Function block in the model.

More About

- “About Embedded Target Development” on page 37-2

- “Support Toolchain Approach with Custom Target” on page 37-81
- “Support Model Referencing” on page 37-83
- “Support Compiler Optimization Level Control” on page 37-95
- “Support C Function Prototype Control” on page 37-97
- “Support C++ Class Interface Control” on page 37-100
- “Support Concurrent Execution of Multiple Tasks” on page 37-102

Customize Template Makefiles

To configure or customize a template makefile (TMF), you should be familiar with how the `make` command works and how it processes makefiles. You should also understand makefile build rules. For information on these topics, refer to the documentation provided with the `make` utility you use.

In this section...

“Template Makefiles and Tokens” on page 37-62

“Invoke the `make` Utility” on page 37-68

“Structure of the Template Makefile” on page 37-69

“Customize and Create Template Makefiles” on page 37-73

Template Makefiles and Tokens

TMFs are made up of statements containing tokens. The build process expands tokens and creates a makefile, `model.mk`. TMFs are designed to generate makefiles for specific compilers on specific platforms. The generated `model.mk` file is tailored to compile and link code generated from your model, using commands specific to your development system.



Creation of `model.mk`

Template Makefile Tokens

The `make_rtw` command (or a different command provided with some targets) directs the process of generating `model.mk`. The `make_rtw` command processes the TMF specified on the **Code Generation** pane of the Configuration Parameters dialog box. `make_rtw` copies the TMF, line by line, expanding each token encountered. Template Makefile Tokens Expanded by `make_rtw` lists the tokens and their expansions.

These tokens are used in several ways by the expanded makefile:

- To control the conditional behavior in the makefile. The conditionals are used to control the source file lists, library names, target to be built, and other build-related information.
- To provide the macro definitions for compiling the files, for example, `-DINTEGER_CODE=1`.

Template Makefile Tokens Expanded by `make_rtw`

Token	Expansion
General purpose	
>ADDITIONAL_LDFLAGS<	Linker flags automatically added by blocks.
>ALT_MATLAB_BIN<	Alternate full pathname for the MATLAB executable; value is different than value for MATLAB_BIN token when the full pathname contains spaces.
>ALT_MATLAB_ROOT<	Alternate full pathname for the MATLAB installation; value is different than value for MATLAB_ROOT token when the full pathname contains spaces.
>BUILDDARGS<	Options passed to <code>make_rtw</code> . This token is provided so that the contents of your <code>model.mk</code> file changes when you change the build arguments, thus forcing an update of modules when your build options change.
>COMBINE_OUTPUT_UPDATE_FCNS<	True (1) when Single output/update function is selected, otherwise False (0). Used for the macro definition <code>-DNESTEPFCN=1</code> .
>COMPUTER<	Computer type. See the MATLAB computer command.
>EXPAND_LIBRARY_LOCATION<	Location of precompiled library file. The <code>TargetPreCompLibLocation</code> configuration parameter can override this setting. For examples, see “Control Library Location and Naming During Build” on page 35-7.
>EXPAND_LIBRARY_NAME<	Library name. For examples, see “Control Library Location and Naming During Build” on page 35-7 and “Modify the Template Makefile for <code>rtwmakecfg</code> ” on page 35-27.

Token	Expansion
>EXPAND_LIBRARY_SUFFIX<	Library suffix. The <code>TargetLibSuffix</code> configuration parameter can override this setting. For examples, see “Control Library Location and Naming During Build” on page 35-7.
>EXT_MODE<	True (1) to enable generation of external mode support code, otherwise False (0).
>EXTMODE_TRANSPORT<	Index of transport mechanism (for example, <code>tcpip</code> , <code>serial</code>) for external mode.
>EXTMODE_STATIC<	True (1) if static memory allocation is selected for external mode. False (0) if dynamic memory allocation is selected.
>EXTMODE_STATIC_SIZE<	Size of static memory allocation buffer (if any) for external mode.
>GENERATE_ERT_S_FUNCTION<	True (1) when Create SIL block is selected, otherwise False (0). Used for control of the makefile target of the build.
>INCLUDE_MDL_TERMINATE_FCN<	True (1) when Terminate function required is selected, otherwise False (0). Used for the macro definition <code>-DTERMFCN==1</code> .
>INTEGER_CODE<	True (1) when Support floating-point numbers is not selected, otherwise False (0). <code>INTEGER_CODE</code> is a required macro definition when compiling the source code and is used when selecting precompiled libraries to link against.
>MAKEFILE_NAME<	<code>model.mk</code> — The name of the makefile that was created from the TMF.
>MAT_FILE<	True (1) when MAT-file logging is selected, otherwise False (0). <code>MAT_FILE</code> is a required macro definition when compiling the source code and also is used to include logging code in the build process.
>MATLAB_BIN<	Location of the MATLAB executable.
>MATLAB_ROOT<	Path to where MATLAB is installed.

Token	Expansion
>MEM_ALLOC<	Either RT_MALLOC or RT_STATIC. Indicates how memory is to be allocated.
>MEXEXT<	MEX-file extension. See the MATLAB mexext command.
>MODEL_MODULES<	Additional generated source modules. For example, you can split a large model into two files, <i>model.c</i> and <i>model1.c</i> . In this case, this token expands to <i>model1.c</i> .
>MODEL_MODULES_OBJ<	Object filenames (.obj) corresponding to additional generated source modules.
>MODEL_NAME<	Name of the Simulink block diagram currently being built.
>MULTITASKING<	True (1) if solver mode is multitasking, otherwise False (0).
>NCSTATES<	Number of continuous states.
>NUMST<	Number of sample times in the model.
>PORTABLE_WORDSIZES<	True (1) when Enable portable word sizes is selected, otherwise False (0).
>RELEASE_VERSION<	The MATLAB release version.
>S_FUNCTIONS<	List of noninlined S-function sources.
>S_FUNCTIONS_LIB<	List of S-function libraries available for linking.
>S_FUNCTIONS_OBJ<	Object (.obj) file list corresponding to noninlined S-function sources.
>SOLVER<	Solver source filename, for example, ode3.c.
>SOLVER_OBJ<	Solver object (.obj) filename, for example, ode3.obj.
>TARGET_LANG_EXT<	c when the Language selection is C, cpp when the Language selection is C++. Used in the makefile to control the extension on generated source files.
>TGT_FCN_LIB<	Specifies compiler command line options. The line in the makefile is TGT_FCN_LIB = >TGT_FCN_LIB< . Use this token in a makefile conditional statement to

Token	Expansion										
	<p>specify a standard math library as a compiler option. Possible <code>>TGT_FCN_LIB<</code> token values are:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="background-color: #cccccc;">Value</th> <th style="background-color: #cccccc;">Generates Calls To</th> </tr> </thead> <tbody> <tr> <td>Name of custom CRL</td> <td>ISO®/IEC 9899:1990 C (ANSI_C) standard math library</td> </tr> <tr> <td>ISO_C</td> <td>ISO/IEC 9899:1999 C standard math library</td> </tr> <tr> <td>ISO_C++</td> <td>ISO/IEC 14882:2003 C++ standard math library</td> </tr> <tr> <td>GNU</td> <td>GNU extensions to the ISO/IEC 9899:1999 C standard math library</td> </tr> </tbody> </table>	Value	Generates Calls To	Name of custom CRL	ISO®/IEC 9899:1990 C (ANSI_C) standard math library	ISO_C	ISO/IEC 9899:1999 C standard math library	ISO_C++	ISO/IEC 14882:2003 C++ standard math library	GNU	GNU extensions to the ISO/IEC 9899:1999 C standard math library
Value	Generates Calls To										
Name of custom CRL	ISO®/IEC 9899:1990 C (ANSI_C) standard math library										
ISO_C	ISO/IEC 9899:1999 C standard math library										
ISO_C++	ISO/IEC 14882:2003 C++ standard math library										
GNU	GNU extensions to the ISO/IEC 9899:1999 C standard math library										
<code>>TID01EQ<</code>	True (1) if sampling rates of the continuous task and the first discrete task are equal, otherwise False (0).										
S-function and build information support											
<p>Note: For examples of the tokens in this section, see “Modify the Template Makefile for <code>rtwmakecfg</code>” on page 35-27.</p>											
<code>>START_EXPAND_INCLUDES<</code> <code>>EXPAND_DIR_NAME<</code> <code>>END_EXPAND_INCLUDES<</code>	List of folder names to add to the include path. Additionally, the <code>ADD_INCLUDES</code> macro must be added to the <code>INCLUDES</code> line.										
<code>>START_EXPAND_LIBRARIES<</code> <code>>EXPAND_LIBRARY_NAME<</code> <code>>END_EXPAND_LIBRARIES<</code>	List of library names.										
<code>>START_EXPAND_MODULES<</code> <code>>EXPAND_MODULE_NAME<</code> <code>>END_EXPAND_MODULES<</code>	Library module names within <code>>START_EXPAND_LIBRARIES<</code> and <code>>START_PRECOMP_LIBRARIES<</code> library lists.										
<code>>START_EXPAND_RULES<</code> <code>>EXPAND_DIR_NAME<</code> <code>>END_EXPAND_RULES<</code>	Makefile rules.										
<code>>START_PRECOMP_LIBRARIES<</code>	List of precompiled library names.										

Token	Expansion
>EXPAND_LIBRARY_NAME< >END_PRECOMP_LIBRARIES<	
Model reference support	
Note: For examples of the tokens in this section, see “Providing Model Referencing Support in the TMF” on page 37-85.	
>MASTER_ANCHOR_DIR<	For parallel builds, current work folder (<code>pwd</code>) at the time the build started.
>MODELLIB<	Name of the library file generated for the current model.
>MODELREFS<	List of models referenced by the top model.
>MODELREF_LINK_LIBS<	List of referenced model libraries against which the top model links.
>MODELREF_LINK_RSPFILE_NAME<	Name of a response file against which the top model links. This token is valid only for build environments that support linker response files. For an example of its use, see <code>matlabroot/rtw/c/grt/grt_vc.tmf</code> .
>MODELREF_TARGET_TYPE<	Type of target being built. Possible values are <ul style="list-style-type: none"> • NONE: Standalone model or top model referencing other models • RTW: Model reference target build • SIM: Model reference simulation target build
>RELATIVE_PATH_TO_ANCHOR<	Relative path, from the location of the generated makefile, to the MATLAB working folder.
>START_DIR<	Current work folder (<code>pwd</code>) at the time the build started. This token is required for parallel builds.
>START_MDLREFINC_EXPAND_INCLUDES< >MODELREF_INC_PATH< >END_MDLREFINC_EXPAND_INCLUDES<	List of include paths for models referenced by the top model.
>SHARED_BIN_DIR<	Folder for the library file built from the shared source files.

Token	Expansion
>SHARED_LIB<	Library file built from the shared source files, including the path to the library folder.
>SHARED_SRC<	Shared source files specification, including the path to the shared utilities folder.
>SHARED_SRC_DIR<	Folder for shared source files.

These tokens are expanded by substitution of parameter values known to the build process. For example, if the source model contains blocks with two different sample times, the TMF statement

```
NUMST = |>NUMST<|
```

expands to the following in *model.mk*.

```
NUMST = 2
```

In addition to the above, `make_rtw` expands tokens from other sources:

- Target-specific tokens defined in the target options of the Configuration Parameters dialog box
- Structures in the `rtwoptions` section of the system target file. Structures in the `rtwoptions` structure array that contain the field `makevariable` are expanded.

The following example is extracted from `matlabroot/rtw/c/grt/grt.tlc`. The section starting with `BEGIN_RTW_OPTIONS` contains MATLAB code that sets up `rtwoptions`. The following directive causes the `|>EXT_MODE<|` token to be expanded to 1 (on) or 0 (off), depending on how you set the external mode options.

```
rtwoptions(2).makevariable = 'EXT_MODE'
```

Invoke the make Utility

- “make Command” on page 37-68
- “make Utility Versions” on page 37-69

make Command

After creating *model.mk* from your TMF, the build process invokes a `make` command. To invoke `make`, the build process issues this command.

```
makecommand -f model.mk
```

makecommand is defined by the MAKECMD macro in your target's TMF (see “Structure of the Template Makefile” on page 37-69). You can specify additional options to `make` in the **Make command** field of the **Code Generation** pane. (See the sections “Specify a Make Command” on page 25-21 and “Template Makefiles and Make Options” on page 17-26.)

For example, specifying `OPT_OPTS=-O2` in the **Make command** field causes `make_rtw` to generate the following `make` command.

```
makecommand -f model.mk OPT_OPTS=-O2
```

A comment at the top of the TMF specifies the available `make` command options. If these options do not provide you with enough flexibility, you can configure your own TMF.

make Utility Versions

The `make` utility lets you control nearly every aspect of building your real-time program. There are several different versions of `make` available. The code generator provides the Free Software Foundation GNU `make` for both UNIX⁷ and PC platforms in platform-specific subfolders under

```
matlabroot/bin
```

It is possible to use other versions of `make` with the code generator, although GNU Make is recommended. To be compatible with the code generator, verify that your version of `make` supports the following command format.

```
makecommand -f model.mk
```

Structure of the Template Makefile

A TMF has multiple sections, including the following:

- **Abstract** — Describes what the makefile targets. Here is a representative abstract from the GRT TMFs in *matlabroot/rtw/c/grt* (open):

```
# File      : grt_lcc.tmf
#
# Abstract:
#   Template makefile for building a PC-based stand-alone generic real-time
#   version of Simulink model using generated C code and LCC compiler
#   Version 2.4.
#
```

7. UNIX is a registered trademark of The Open Group in the United States and other countries.

```

# This makefile attempts to conform to the guidelines specified in the
# IEEE Std 1003.2-1992 (POSIX) standard. It is designed to be used
# with GNU Make (gmake) which is located in matlabroot/bin/win32.
#
# Note that this template is automatically customized by the build
# procedure to create "<model>.mk"
#
# The following defines can be used to modify the behavior of the
# build:
#   OPT_OPTS      - Optimization options. Default is none. To enable
#                   debugging specify as OPT_OPTS=-g4.
#   OPTS          - User specific compile options.
#   USER_SRCS    - Additional user sources, such as files needed by
#                   S-functions.
#   USER_INCLUDES - Additional include paths
#                   (i.e. USER_INCLUDES="-Iwhere-ever -Iwhere-ever2")
#                   (For Lcc, have a '/' as file separator before the
#                   file name instead of a '\'.
#                   i.e., d:\work\proj1\myfile.c - reqd for 'gmake')
#
# This template makefile is designed to be used with a system target
# file that contains 'rtwgensettings.BuildDirSuffix'. See grt.tlc.

```

- **Macros read by make_rtw section** — Defines macros that tell `make_rtw` how to process the TMF. Here is a representative **Macros read by make_rtw** section from the GRT TMFs in `matlabroot/rtw/c/grt` (open):

```

#----- Macros read by make_rtw -----
#
# The following macros are read by the build procedure:
#
# MAKECMD      - The command that invokes the make utility
# HOST         - The platform (for example, PC) for which this TMF is written
# SHELL        - An operating system shell command (for example, cmd) for the platform
# BUILD        - The flag that indicates whether to invoke make from the build procedure
# SYS_TARGET_FILE - Name of system target file.

MAKECMD      = "%MATLAB%\bin\win32\gmake"
HOST         = PC
SHELL        = cmd
BUILD        = yes
SYS_TARGET_FILE = grt.tlc
BUILD_SUCCESS = *** Created
COMPILER_TOOL_CHAIN = lcc

MAKEFILE_FILESEP = /

```

The macros in this section might include:

- **MAKECMD** — Specifies the command used to invoke the make utility. For example, if `MAKECMD = mymake`, then the make command invoked is

```
mymake -f model.mk
```

- **HOST** — Specifies the platform targeted by this TMF. This can be PC, UNIX, *computer_name* (see the MATLAB `computer` command), or ANY.
- **SHELL** — Specifies an operating system shell command for the platform. For Windows, this can be `cmd`.
- **BUILD** — Instructs `make_rtw` whether or not it should invoke `make` from the build procedure. Specify `yes` or `no`.
- **SYS_TARGET_FILE** — Specifies the name of the system target file or the value `any`. This is used for consistency checking by `make_rtw` to verify the system target file specified in the **Target selection** panel of the **Code Generation** pane of the Configuration Parameters dialog box. If you specify `any`, you can use the TMF with any system target file.
- **BUILD_SUCCESS** — Optional macro that specifies the build success message to be displayed for `make` completion on the PC. For example,

```
BUILD_SUCCESS = ### Successful creation of
```

The **BUILD_SUCCESS** macro, if used, replaces the standard build success message found in the TMFs distributed with the bundled code generator targets (such as GRT):

```
@echo ### Created executable $(MODEL).exe
```

Your TMF must include either the standard build success message, or use the **BUILD_SUCCESS** macro. For an example of the use of **BUILD_SUCCESS**, see `matlabroot/rtw/c/grt/grt_lcc.tmf` or the code example above this list of macros.

- **BUILD_ERROR** — Optional macro that specifies the build error message to be displayed when an error is encountered during the `make` procedure. For example,

```
BUILD_ERROR = ['Error while building ', modelName]
```

- **VERBOSE_BUILD_OFF_TREATMENT = PRINT_OUTPUT_ALWAYS** — Optional macro to include if you want the makefile output to be displayed regardless of the setting of the **Verbose build** option in the **Code Generation > Debug** pane.
- **COMPILER_TOOL_CHAIN** — For builds on Windows systems, specifies which compiler setup file — located in `matlabroot/toolbox/rtw/rtw` (open) — to use:
 - `lcc` selects `setup_for_lcc.m`
 - `vc` selects `setup_for_visual.m`

- `vcx64` selects `setup_for_visual_x64.m`
- `default` selects `setup_for_default.m`

For builds on UNIX systems, specify `unix`. Other values are flagged as unknown and `make_rtw` uses `setup_for_default.m`.

Note: Do not omit `COMPILER_TOOL_CHAIN` or leave it unspecified in your TMF. If your compiler is not the host compiler, specify `COMPILER_TOOL_CHAIN = default`.

- **DOWNLOAD** — An optional macro that you can specify as yes or no. If specified as yes (and `BUILD=yes`), then `make` is invoked a second time with the download target.

```
make -f model.mk download
```

- **DOWNLOAD_SUCCESS** — An optional macro that you can use to specify the download success message to be used when looking for a completed download. For example,

```
DOWNLOAD_SUCCESS = ### Downloaded
```

- **DOWNLOAD_ERROR** — An optional macro that you can use to specify the download error message to be displayed when an error is encountered during the download. For example,

```
DOWNLOAD_ERROR = ['Error while downloading ', modelName]
```

- **Tokens expanded by `make_rtw` section** — Defines the tokens that `make_rtw` expands. Here is a brief excerpt from a representative **Tokens expanded by `make_rtw` section** from the GRT TMFs in `matlabroot/rtw/c/grt` (open):

```
#----- Tokens expanded by make_rtw -----
#
# The following tokens, when wrapped with ">" and "<" are expanded by the
# build procedure.
#
# MODEL_NAME           - Name of the Simulink block diagram
# MODEL_MODULES        - Any additional generated source modules
# MAKEFILE_NAME        - Name of makefile created from template makefile <model>.mk
# MATLAB_ROOT          - Path to where MATLAB is installed.
...

MODEL                 = >MODEL_NAME<|
MODULES               = >MODEL_MODULES<|
MAKEFILE              = >MAKEFILE_NAME<|
```

```
MATLAB_ROOT      = |>MATLAB_ROOT<|
...
```

For more information about TMF tokens, see [Template Makefile Tokens Expanded by make_rtw](#).

- Subsequent sections vary based on compiler, host, and target. Some common sections include `Model` and reference models, `External mode`, `Tool Specifications` or `Tool Definitions`, `Include Path`, `C Flags`, `Additional Libraries`, and `Source Files`.
- `Rules` section — Contains the make rules used in building an executable from the generated source code. The build rules are typically specific to your version of make. The `Rules` section might be followed by related sections such as `Dependencies`.

Customize and Create Template Makefiles

- “Introduction” on page 37-73
- “Setting Up a Template Makefile” on page 37-73
- “Using Macros and Pattern Matching Expressions in a Template Makefile” on page 37-74
- “Customizing Generated Makefiles with `rtwmakecfg`” on page 37-76
- “Supporting Continuous Time in Custom Targets” on page 37-76
- “Model Reference Considerations” on page 37-77

Introduction

This section describes the mechanics of setting up a custom template makefile (TMF) and incorporating it into the build process. It also discusses techniques for modifying a TMF and MATLAB file mechanisms associated with the TMF.

Before creating a custom TMF, you should read “Folder and File Naming Conventions” on page 37-11 to understand the folder structure and MATLAB path requirements for custom targets.

Setting Up a Template Makefile

To customize or create a new TMF, you should copy an existing GRT or ERT TMF from one of the following locations:

```
matlabroot/rtw/c/grt (open)
matlabroot/rtw/c/ert (open)
```

Place the copy in the same folder as the associated system target file (STF). Usually, this is the `mytarget/mytarget` folder within the target folder structure. Then, rename your TMF (for example, `mytarget.tmf`) and modify it.

To allow the build process to locate and select your TMF, you must provide information in the STF file header (see “System Target File Structure” on page 37-30). For a target that implements a single TMF, the standard way to specify the TMF to be used in the build process is to use the TMF directive of the STF file header.

```
TMF: mytarget.tmf
```

Using Macros and Pattern Matching Expressions in a Template Makefile

This section shows, through an example, how to use macros and file-pattern-matching expressions in a TMF to generate commands in the `model.mk` file.

The `make` utility processes the `model.mk` makefile and generates a set of commands based upon dependency rules defined in `model.mk`. After `make` generates the set of commands for building or rebuilding `test`, `make` executes them.

For example, to build a program called `test`, `make` must link the object files. However, if the object files don't exist or are out of date, `make` must compile the source code. Thus there is a dependency between source and object files.

Each version of `make` differs slightly in its features and how rules are defined. For example, consider a program called `test` that gets created from two sources, `file1.c` and `file2.c`. Using most versions of `make`, the dependency rules would be

```
test: file1.o file2.o
    cc -o test file1.o file2.o

file1.o: file1.c
    cc -c file1.c

file2.o: file2.c
    cc -c file2.c
```

In this example, a UNIX⁸ environment is assumed. In a PC environment the file extensions and compile and link commands are different.

In processing the first rule

8. UNIX is a registered trademark of The Open Group in the United States and other countries.


```
test: file1.o file2.o
```

make sees that to build `test`, it needs to build `file1.o` and `file2.o`. To build `file1.o`, make processes the rule

```
file1.o: file1.c
```

If `file1.o` doesn't exist, or if `file1.o` is older than `file1.c`, make compiles `file1.c`.

The format of TMFs follows the above example. Our TMFs use additional features of make such as macros and file-pattern-matching expressions. In most versions of make, a macro is defined with

```
MACRO_NAME = value
```

References to macros are made with `$(MACRO_NAME)`. When make sees this form of expression, it substitutes `value` for `$(MACRO_NAME)`.

You can use pattern matching expressions to make the dependency rules more general. For example, using GNU⁹ Make, you could replace the two “`file1.o: file1.c`” and “`file2.o: file2.c`” rules with the single rule

```
%.o : %.c
      cc -c $<
```

Note that `$<` in the previous example is a special macro that equates to the dependency file (that is, `file1.c` or `file2.c`). Thus, using macros and the “%” pattern matching character, the previous example can be reduced to

```
SRCS = file1.c file2.c
OBJS = $(SRCS:.c=.o)
```

```
test: $(OBJS)
      cc -o $@ $(OBJS)
```

```
%.o : %.c
      cc -c $<
```

Note that the `$@` macro above is another special macro that equates to the name of the current dependency target, in this case `test`.

This example generates the list of objects (`OBJS`) from the list of sources (`SRCS`) by using the text substitution feature for macro expansion. It replaces the source file extension

9. GNU is a registered trademark of the Free Software Foundation.

(for example, `.c`) with the object file extension (`.o`). This example also generalized the build rule for the program, `test`, to use the special "\$@" macro.

Customizing Generated Makefiles with `rtwmakecfg`

TMFs provide tokens that let you add the following items to generated makefiles:

- Source folders
- Include folders
- Run-time library names
- Run-time module objects

S-functions can add this information to the makefile by using an `rtwmakecfg.m` file function. This function is particularly useful when building a model that contains one or more of your S-Function blocks, such as device driver blocks.

To add information pertaining to an S-function to the makefile,

- 1 Create the function `rtwmakecfg` in file `rtwmakecfg.m`. The code generator associates this file with your S-function based on its folder location.
- 2 Modify your target's TMF such that it supports macro expansion for the information returned by `rtwmakecfg` functions.

After the TLC phase of the build process, when generating a makefile from the TMF, the build process searches for an `rtwmakecfg.m` file in the folder that contains the S-function component. If it finds the file, the build process calls the `rtwmakecfg` function. For more information, see “Use `rtwmakecfg.m` API to Customize Generated Makefiles” on page 35-24.

Supporting Continuous Time in Custom Targets

If you want your custom ERT-based target to support continuous time, you must update your template makefile (TMF) and the static main program module (for example, `mytarget_main.c`) for your target.

Template Makefile Modifications

Add the `NCSTATES` token expansion after the `NUMST` token expansion, as follows:

```
NUMST = |>NUMST<|  
NCSTATES = |>NCSTATES<|
```

In addition, add `NCSTATES` to the `CPP_REQ_DEFINES` macro, as in the following example:

```
CPP_REQ_DEFINES = -DMODEL=$(MODEL) -DNUMST=$(NUMST) -DNCSTATES=$(NCSTATES) \
-DMAT_FILE=$(MAT_FILE)
-DINTEGER_CODE=$(INTEGER_CODE) \
-DONESTEPFCN=$(ONESTEPFCN) -DTERMFCN=$(TERMFCN) \
-DHAVESTDIO
-DMULTI_INSTANCE_CODE=$(MULTI_INSTANCE_CODE) \
```

Modifications to Main Program Module

The main program module defines a static main function that manages task scheduling for the supported tasking modes of single- and multiple-rate models. `NUMST` (the number of sample times in the model) determines whether the main function calls multirate or single-rate code. However, when a model uses continuous time, do not rely on `NUMST` directly.

When the model has continuous time and the flag `TID01EQ` is true, both continuous time and the fastest discrete time are treated as one rate in generated code. The code associated with the fastest discrete rate is guarded by a major time step check. When the model has only two rates, and `TID01EQ` is true, the generated code has a single-rate call interface.

To support models that have continuous time, update the static main module to take `TID01EQ` into account, as follows:

- 1 Before `NUMST` is referenced in the file, add the following code:

```
#if defined(TID01EQ) && TID01EQ == 1 && NCSTATES == 0
#define DISC_NUMST (NUMST - 1)
#else
#define DISC_NUMST NUMST
#endif
```

- 2 Replace instances of `NUMST` in the file by `DISC_NUMST`.

Model Reference Considerations

See “Support Model Referencing” on page 37-83 for important information on TMF modifications you may need to make to support the code generator model referencing features.

Note: If you are using a TMF without the variables `SHARED_SRC` or `MODELREFS`, the file might have been used with a previous release of Simulink software. If you want your

TMF to support model referencing, add either variable SHARED_SRC or MODELREFS to the make file.

More About

- “About Embedded Target Development” on page 37-2
- “Sample Custom Targets” on page 37-9
- “Customize System Target Files” on page 37-29
- “Custom Target Optional Features” on page 37-79
- “Support Toolchain Approach with Custom Target” on page 37-81

Custom Target Optional Features

This section describes how to configure a custom embedded target to support these optional features.

To ...	Use Target Configuration Parameters ...	For more information, see ...
Indicate a custom target is toolchain-compliant	UseToolchainInfoCompliantGenerateMakefile	“Support Toolchain Approach with Custom Target” on page 37-81
Build a model that includes referenced models and uses a custom target	ModelReferenceCompliantParMdlRefBuildCompliant (parallel build support)	“Support Model Referencing” on page 37-83
Control the compiler optimization level building generated code for a custom target	CompOptLevelCompliant	“Support Compiler Optimization Level Control” on page 37-95
Control C function prototypes of initialize and step functions that are generated for a model that uses a custom target	ModelStepFunctionPrototypeCompliant (ERT only)	“Support C Function Prototype Control” on page 37-97
Control C++ class interfaces that are generated for a model that uses a custom target	CPPClassGenCompliant (ERT only)	“Support C++ Class Interface Control” on page 37-100
Enable concurrent execution of multiple tasks on a multicore platform for a model that uses a custom target	ConcurrentExecutionCompliant	“Support Concurrent Execution of Multiple Tasks” on page 37-102

The required configuration changes are modifications to your system target file (STF), and in some cases also modifications to your template makefile (TMF) or your custom static main program.

The API for STF callbacks provides a function `SelectCallback` for use in STFs. `SelectCallback` is associated with the target rather than with its individual options. If you implement a `SelectCallback` function for a target, it is triggered whenever the user selects the target in the System Target File Browser.

The API provides the functions `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` for controlling custom target configuration options from a user-written `SelectCallback` function. (For function descriptions and examples, see the function reference pages.)

The general requirements for supporting one of the optional features include:

- To support model referencing or compiler optimization level control, the target must be derived from the GRT or the ERT target. To support C function prototype control or C++ class interface control, the target must be derived from the ERT target.
- The system target file (STF) must declare feature compliance by including one of the target configuration parameters listed above in a `SelectCallback` function call.
- Additional changes such as TMF modifications or static main program modifications may be required, depending on the feature. See the detailed steps in the subsections for individual features.

For an example that shows how to configure custom target optional features, see “Customize System Target Files” on page 37-29.

More About

- “About Embedded Target Development” on page 37-2
- “Sample Custom Targets” on page 37-9
- “Support Toolchain Approach with Custom Target” on page 37-81
- “Support Model Referencing” on page 37-83
- “Support Compiler Optimization Level Control” on page 37-95
- “Support C Function Prototype Control” on page 37-97
- “Support C++ Class Interface Control” on page 37-100
- “Support Concurrent Execution of Multiple Tasks” on page 37-102

Support Toolchain Approach with Custom Target

This section describes how to configure a custom system target file to support builds with the toolchain approach.

In the Configuration Parameters dialog box, on the Code Generation pane of, you can present either the build controls for the toolchain approach or the template makefile approach. The model parameters that contribute to determining which build controls appear include these parameters.

Model Parameter	Value	Notes
UseToolchainInfoCompliant	on	For toolchain approach, set this parameter to 'on'. For TMF approach, set this parameter to 'off'.
GenerateMakefile	on	For toolchain approach, set this parameter to 'on'.

When the dialog box detects that the selected target has these properties, the dialog box recognizes the target as toolchain-compliant and displays the build controls for the toolchain approach.

Because the custom target file cannot set these properties directly, use a `SelectCallback` function in the custom target file to set the properties. The `SelectCallback` function call in the `RTW_OPTION` section of the TLC file can take the form:

```
rtwgensettings.SelectCallback = 'enableToolchainCompliant(hSrc, hDlg)';
```

A corresponding callback function can contain:

```
function enableToolchainCompliant(hSrc, hDlg)
    hCS = hSrc.getConfigSet();

    % The following parameters enable toolchain compliance.
    slConfigUISetVal(hDlg, hSrc, 'UseToolchainInfoCompliant', 'on');
    hCS.setProp('GenerateMakefile', 'on');

    % The following parameters are not required for toolchain compliance.
    % But, it is recommended practice to set these default values and
    % disable the parameters (as shown).
    hCS.setProp('RTWCompilerOptimization', 'off');
    hCS.setProp('MakeCommand', 'make_rtw');
    hCS.setPropEnabled('RTWCompilerOptimization', false);
```

```
hCS.setPropEnabled('MakeCommand',false);  
end
```

When you select your custom target, the configuration parameters dialog box displays the toolchain approach build controls. For an example, see “Create a Custom Target Configuration” on page 37-48.

For an example that shows how to configure custom target optional features, see “Customize System Target Files” on page 37-29.

More About

- “Customize System Target Files” on page 37-29
- “Support Model Referencing” on page 37-83
- “Support Compiler Optimization Level Control” on page 37-95
- “Support C Function Prototype Control” on page 37-97
- “Support C++ Class Interface Control” on page 37-100
- “Support Concurrent Execution of Multiple Tasks” on page 37-102

Support Model Referencing

This section describes how to configure a custom embedded target to support model referencing. Without the described modifications, you will not be able to use the custom target when building a model that includes referenced models. If you do not intend to use referenced models with your target, you can skip this section. If you later find that you need to use referenced models, you can upgrade your target then.

In this section...

- “About Model Referencing with a Custom Target” on page 37-83
- “Declaring Model Referencing Compliance” on page 37-84
- “Providing Model Referencing Support in the TMF” on page 37-85
- “Controlling Configuration Option Value Agreement” on page 37-88
- “Supporting the Shared Utilities Folder” on page 37-88
- “Verifying Worker Configuration for Parallel Builds of Model Reference Hierarchies (Optional)” on page 37-92
- “Preventing Resource Conflicts (Optional)” on page 37-94

About Model Referencing with a Custom Target

The requirements for supporting model referencing are as follows:

- The target must be derived from the GRT target or the ERT target.
- The system target file (STF) must declare model reference compliance, as described in “Declaring Model Referencing Compliance” on page 37-84.
- The template makefile (TMF) must define some entities that support model referencing, as described in “Providing Model Referencing Support in the TMF” on page 37-85.
- The TMF must support using the Shared Utilities folder, as described in “Supporting the Shared Utilities Folder” on page 37-88.

Optionally, you can provide additional capabilities that support model referencing:

- You can configure a target to support parallel builds for large model reference hierarchies (see “Reduce Build Time for Referenced Models” on page 25-45). To do

this, you must modify the STF and TMF for parallel builds as described in “Declaring Model Referencing Compliance” on page 37-84 and “Providing Model Referencing Support in the TMF” on page 37-85.

- If your target supports parallel builds for large model reference hierarchies, you can additionally set up automatic verification of MATLAB Distributed Computing Server workers, as described in “Verifying Worker Configuration for Parallel Builds of Model Reference Hierarchies (Optional)” on page 37-92.
- You can modify hook files to handle referenced models differently than top models to prevent resource conflicts, as described in “Preventing Resource Conflicts (Optional)” on page 37-94.

See “Overview of Model Referencing” for information about model referencing in Simulink models, and “Generate Code for Referenced Models” on page 8-4 for information about model referencing in generated code.

For an example that shows how to configure custom target optional features, see “Customize System Target Files” on page 37-29.

Declaring Model Referencing Compliance

To declare model reference compliance for your target, you must implement a callback function that sets the `ModelReferenceCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `ModelReferenceCompliant` flag as follows:

```
siConfigUISetVal(hDlg,hSrc,'ModelReferenceCompliant','on');  
siConfigUISetEnabled(hDlg,hSrc,'ModelReferenceCompliant',false);
```

If you might use the target to build models containing large model reference hierarchies, consider configuring the target to support parallel builds, as discussed in “Reduce Build Time for Referenced Models” on page 25-45.

To configure a target for parallel builds, your callback function must also set the `ParMdlRefBuildCompliant` flag as follows:

```
sIConfigUISetVal(hDlG,hSrc,'ParMdlRefBuildCompliant','on');
sIConfigUISetEnabled(hDlG,hSrc,'ParMdlRefBuildCompliant',false);
```

For more information about the STF callback API, see the `sIConfigUISetVal`, `sIConfigUISetEnabled`, and `sIConfigUISetVal` function reference pages.

Providing Model Referencing Support in the TMF

Do the following to configure the template makefile (TMF) to support model referencing:

- 1 Add the following make variables and tokens to be expanded when the makefile is generated:

```
MODELREFS           = |>MODELREFS<|
MODELLIB            = |>MODELLIB<|
MODELREF_LINK_LIBS  = |>MODELREF_LINK_LIBS<|
MODELREF_LINK_RSPFILE = |>MODELREF_LINK_RSPFILE_NAME<|
MODELREF_INC_PATH   = |>START_MDLREFINC_EXPAND_INCLUDES<|\
  -I|>MODELREF_INC_PATH<| |>END_MDLREFINC_EXPAND_INCLUDES<|
RELATIVE_PATH_TO_ANCHOR = |>RELATIVE_PATH_TO_ANCHOR<|
MODELREF_TARGET_TYPE = |>MODELREF_TARGET_TYPE<|
```

The following code excerpt shows how makefile tokens are expanded for a referenced model.

```
MODELREFS           =
MODELLIB            = engine3200cc_rtwlib.a
MODELREF_LINK_LIBS  =
MODELREF_LINK_RSPFILE =
MODELREF_INC_PATH   =
RELATIVE_PATH_TO_ANCHOR = ../../..
MODELREF_TARGET_TYPE = RTW
```

The following code excerpt shows how makefile tokens are expanded for the top model that references the referenced model.

```
MODELREFS           = engine3200cc transmission
MODELLIB            = archlib.a
MODELREF_LINK_LIBS  = engine3200cc_rtwlib.a transmission_rtwlib.a
MODELREF_LINK_RSPFILE =
MODELREF_INC_PATH   = -I../slprj/ert/engine3200cc -I../slprj/ert/transmission
RELATIVE_PATH_TO_ANCHOR = ..
MODELREF_TARGET_TYPE = NONE
```

Token	Expands to
MODELREFS for the top model	List of referenced model names.

Token	Expands to
MODELLIB	Name of the library generated for the model.
MODELREF_LINK_LIBS token for the top model	List of referenced model libraries that the top model links against.
MODELREF_LINK_RSPFILE token for the top model	Name of a response file that the top model links against. This token is valid only for build environments that support linker response files. For an example of its use, see <i>matlabroot/rtw/c/grt/grt_vc.tmf</i> .
MODELREF_INC_PATH token for the top model	Include path to the referenced models.
RELATIVE_PATH_TO_ANCHOR	Relative path, from the location of the generated makefile, to the MATLAB working folder.
MODELREF_TARGET_TYPE	Signifies the type of target being built. Possible values are <ul style="list-style-type: none"> • NONE: Standalone model or top model referencing other models • RTW: Model reference target build • SIM: Model reference simulation target build

If you are configuring your target to support parallel builds, as discussed in “Reduce Build Time for Referenced Models” on page 25-45, you must also add the following token definitions to your TMF:

```
START_DIR = |>START_DIR<|
MASTER_ANCHOR_DIR = |>MASTER_ANCHOR_DIR<|
```

Token	Expands to
START_DIR	Current work folder (pwd) at the time the build started.
MASTER_ANCHOR_DIR	Current work folder (pwd) at the time the build started.

- 2 Add `RELATIVE_PATH_TO_ANCHOR` and `MODELREF_INC_PATH` include paths to the overall `INCLUDES` variable.

```
INCLUDES = -I. -I$(RELATIVE_PATH_TO_ANCHOR) $(MATLAB_INCLUDES) $(ADD_INCLUDES) \
$(USER_INCLUDES) $(MODELREF_INC_PATH) $(SHARED_INCLUDES)
```

- 3 Change the `SRCS` variable in your `TMF` so that it initially lists only common modules. Additional modules are then appended conditionally, as described in the next step. For example, change

```
SRCS = $(MODEL).c $(MODULES) ert_main.c $(ADD_SRCS) $(EXT_SRC)
```

to

```
SRCS = $(MODULES) $(S_FUNCTIONS)
```

- 4 Create variables to define the final target of the makefile. You can remove variables that may have existed for defining the final target. For example, remove

```
PROGRAM = ../$(MODEL)
```

and replace it with

```
ifeq ($(MODELREF_TARGET_TYPE), NONE)
# Top model for RTW
PRODUCT          = $(RELATIVE_PATH_TO_ANCHOR)/$(MODEL)
BIN_SETTING      = $(LD) $(LDFLAGS) -o $(PRODUCT) $(SYSLIBS)
BUILD_PRODUCT_TYPE = "executable"
# ERT based targets
SRCS              += $(MODEL).c ert_main.c $(EXT_SRC)
# GRT based targets
# SRCS              += $(MODEL).c grt_main.c rt_sim.c $(EXT_SRC) $(SOLVER)
else
# sub-model for RTW
PRODUCT          = $(MODELLIB)
BUILD_PRODUCT_TYPE = "library"
endif
```

- 5 Create rules for the final target of the makefile (replace existing final target rules). For example:

```
ifeq ($(MODELREF_TARGET_TYPE), NONE)
# Top model for RTW
$(PRODUCT) : $(OBJS) $(SHARED_LIB) $(LIBS) $(MODELREF_LINK_LIBS)
              $(BIN_SETTING) $(LINK_OBJS) $(MODELREF_LINK_LIBS)
              $(SHARED_LIB) $(LIBS)
              @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
else
# sub-model for RTW
$(PRODUCT) : $(OBJS) $(SHARED_LIB) $(LIBS)
              @rm -f $(MODELLIB)
              $(ar) ruvs $(MODELLIB) $(LINK_OBJS)
```

```

        @echo "### Created $(MODELLIB)"
        @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
    endif

```

- 6 Create a rule to allow referenced models to compile files that reside in the MATLAB working folder (pwd).

```

%.o : $(RELATIVE_PATH_TO_ANCHOR) /%.c
    $(CC) -c $(CFLAGS) $<

```

Note: If you are using a TMF without the variables `SHARED_SRC` or `MODELREFS`, the file might have been used with a previous release of Simulink software. If you want your TMF to support model referencing, add either variable `SHARED_SRC` or `MODELREFS` to the make file.

Controlling Configuration Option Value Agreement

By default, the value of a configuration option defined in the system target file for a TLC-based custom target must be the same in any referenced model and its parent model. To relax this requirement, include the `modelReferenceParameterCheck` field in the `rtwoptions` structure element that defines the configuration option, and set the value of the field to 'off'. For example:

```

rtwoptions(2).prompt      = 'My Custom Parameter';
rtwoptions(2).type       = 'Checkbox';
rtwoptions(2).default    = 'on';
rtwoptions(2).modelReferenceParameterCheck = 'on';
rtwoptions(2).tlcvariable = 'mytlcvariable';
...

```

The configuration option **My Custom Parameter** can differ in a referenced model and its parent model. See “Customize System Target Files” on page 37-29 for information about TLC-based system target files, and `rtwoptions` Structure Fields Summary for a list of `rtwoptions` fields.

Supporting the Shared Utilities Folder

- “Overview” on page 37-89
- “Implementing Shared Utilities Folder Support” on page 37-90

Overview

The makefile used by the build process must support compiling and creating libraries, and so on, from the locations in which the code is generated. Therefore, you need to update your makefile and the model reference build process to support the shared utilities location. The **Shared code placement** options have the following requirements:

- **Auto**
 - Standalone model build — Build files go to the build folder; makefile is not updated.
 - Referenced model or top model build — Use shared utilities folder; makefile requires full model reference support.
- **Shared location**
 - Standalone model build — Use shared utilities folder; makefile requires shared location support.
 - Referenced model or top model build — Use shared utilities folder; makefile requires full model reference support.

The shared utilities folder (`slprj/target/_sharedutils`) typically stores generated utility code that is common between a top model and the models it references. You can also force the build process to use a shared utilities folder for a standalone model. See “Code Generation Folder Structure for Model Reference Targets” on page 8-15 for details.

If you want your target to support compilation of code generated in the shared utilities folder, several updates to your template makefile (TMF) are required. Support for Model Reference builds requires the shared utilities folder. See the preceding sections to learn about additional updates for supporting Model Reference builds.

The exact syntax of the changes can vary due to differences in the make utility and compiler/archiver tools used by your target. The examples below are based on the GNU¹⁰ make utility. You can find the following updated TMF examples for GNU and Microsoft Visual C++ make utilities in the GRT and ERT target folders:

- GRT: `matlabroot/rtw/c/grt` (open)
 - `grt_lcc.tmf`
 - `grt_vc.tmf`

10. GNU is a registered trademark of the Free Software Foundation.

- grt_unix.tmf
- ERT: *matlabroot*/rtw/c/ert (open)
 - ert_lcc.tmf
 - ert_vc.tmf
 - ert_unix.tmf

Use the GRT or ERT examples as a guide to the location, within the TMF, of the changes and additions described below.

Note The ERT-based TMFs contain extra code to handle generation of ERT S-functions and Model Reference simulation targets. Your target does not need to handle these cases.

Implementing Shared Utilities Folder Support

Make the following changes to your TMF to support the shared utilities folder:

- 1 Add the following make variables and tokens to be expanded when the makefile is generated:

```
SHARED_SRC      = |>SHARED_SRC<|
SHARED_SRC_DIR  = |>SHARED_SRC_DIR<|
SHARED_BIN_DIR  = |>SHARED_BIN_DIR<|
SHARED_LIB      = |>SHARED_LIB<|
```

SHARED_SRC specifies the shared utilities folder location and the source files in it. A typical expansion in a makefile is

```
SHARED_SRC      = ../slprj/ert/_sharedutils/*.c
```

SHARED_LIB specifies the library file built from the shared source files, as in the following expansion.

```
SHARED_LIB      = ../slprj/ert/_sharedutils/rtwshared.lib
```

SHARED_SRC_DIR and SHARED_BIN_DIR allow specification of separate folders for shared source files and the library compiled from the source files. In the current release, the TMFs use the same path, as in the following expansions.

```
SHARED_SRC_DIR  = ../slprj/ert/_sharedutils
```



```
SHARED_BIN_DIR = ../slprj/ert/_sharedutils
```

- 2 Set the `SHARED_INCLUDES` variable according to whether shared utilities are in use. Then append it to the overall `INCLUDES` variable.

```
SHARED_INCLUDES =
ifneq ($(SHARED_SRC_DIR),)
SHARED_INCLUDES = -I$(SHARED_SRC_DIR)
endif
```

```
INCLUDES = -I. $(MATLAB_INCLUDES) $(ADD_INCLUDES) \
$(USER_INCLUDES) $(SHARED_INCLUDES)
```

- 3 Update the `SHARED_SRC` variable to list shared files explicitly.

```
SHARED_SRC := $(wildcard $(SHARED_SRC))
```

- 4 Create a `SHARED_OBJS` variable based on `SHARED_SRC`.

```
SHARED_OBJS = $(addsuffix .o, $(basename $(SHARED_SRC)))
```

- 5 Create an `OPTS` (options) variable for compilation of shared utilities.

```
SHARED_OUTPUT_OPTS = -o $@
```

- 6 Provide a rule to compile the shared utility source files.

```
$(SHARED_OBJS) : $(SHARED_BIN_DIR)/%.o : $(SHARED_SRC_DIR)/%.c
$(CC) -c $(CFLAGS) $(SHARED_OUTPUT_OPTS) $<
```

- 7 Provide a rule to create a library of the shared utilities. The following example is based on UNIX¹¹.

```
$(SHARED_LIB) : $(SHARED_OBJS)
@echo "### Creating $@"
ar r $@ $(SHARED_OBJS)
@echo "### Created $@"
```

Note: Depending on your make utility, you may be able to combine Steps 6 and 7 into one rule. For example, `gmake` (used with `ert_unix.tmf`) uses:

```
$(SHARED_LIB) : $(SHARED_SRC)
@echo "### Creating $@"
cd $(SHARED_BIN_DIR); $(CC) -c $(CFLAGS) $(GCC_WALL_FLAG_MAX) $(notdir $?)
ar ruvs $@ $(SHARED_OBJS)
@echo "### $@ Created "
```

11. UNIX is a registered trademark of The Open Group in the United States and other countries.

See this and other examples in the files `ert_vc.tmf`, `ert_lcc.tmf`, and `ert_unix.tmf` located at `matlabroot/rtw/c/ert` (open).

- 8 Add `SHARED_LIB` to the rule that creates the final executable.

```
$(PROGRAM) : $(OBJS) $(LIBS) $(SHARED_LIB)
$(LD) $(LDFLAGS) -o $$@ $(LINK_OBJS) $(LIBS)
$(SHARED_LIB) $(SYSLIBS)
@echo "### Created executable: $(MODEL)"
```

- 9 Remove explicit references to `rt_nonfinite.c` from your TMF. For example, change

```
ADD_SRCS = $(RTWLOG) rt_nonfinite.c
```

to

```
ADD_SRCS = $(RTWLOG)
```

Note If your target interfaces to a development environment that is not makefile based, you must make equivalent changes to provide information to your target compilation environment.

Verifying Worker Configuration for Parallel Builds of Model Reference Hierarchies (Optional)

If your target supports parallel builds for large model reference hierarchies, you can additionally set up automatic verification of MATLAB Distributed Computing Server workers. This addresses the possibility that parallel workers might have different configurations, some of which might not be compatible with a specific target build. For example, the required compiler might not be installed on a worker system.

The code generator provides a programming interface that you can use to automatically check the configuration of parallel workers. If parallel workers are not set up as expected, take action, such as reverting to sequential builds or throwing an error.

To set up automatic verification of workers, you must define a parallel configuration check function named `STF_par_cfg_chk`, where `STF` designates your system target file name. For example, the parallel configuration check function for `ert.tlc` is `ert_par_cfg_chk`.

The general syntax for the function is:

```
function varargout = STF_par_cfg_chk(action,varargin)
```

The number of output and input arguments vary according to the *action* specified, and according to the types of information you choose to coordinate between the client and the workers. The function should support the following general sequence of parallel configuration setup calls, differentiated by the first argument passed in:

Call Syntax	Called on:	Action
<code>cfg = STF_par_cfg_chk('getPreferredCfg');</code>	Client	Return a structure representing the preferred configuration for MATLAB Distributed Computing Server workers.
<code>[tf, cfg] = STF_par_cfg_chk('getWorkerCfg', cfg);</code>	Workers	Each worker is passed the MATLAB Distributed Computing Server client's preferred configuration. Return true if the worker can support the preferred configuration; otherwise return false along with a structure representing a configuration the worker can support. Information returned by each worker is added to a cell array of configurations.
<code>[tf, cfg] = STF_par_cfg_chk('getCommonCfg', cfgs);</code>	Client	The client is passed the cell array of worker configurations. If a usable common configuration exists, return true, and return the common configuration to set for all systems. If a common configuration cannot be established, return false or take some action, such as reverting to sequential builds or throwing an error.
<code>tf = STF_par_cfg_chk('setCommonCfg', cfg);</code>	Workers and client	Each system is passed the common configuration to use. Set up the common configuration and, if successful, return true. If errors or issues occur, return false or take some action, such as reverting to sequential builds or throwing an error.
<code>STF_par_cfg_chk('clearCfg');</code>	Workers and client	Clean up after completion of the parallel build.

The parallel configuration check functions for MathWorks provided targets are implemented as wrapper functions that call a function named `parallelMdlRefHostConfigCheckFcn`. For example, see the ERT parallel configuration check function in the file `matlabroot/toolbox/rtw/rtw/`

`ert_par_cfg_chk.m`, and the function it calls in the file `matlabroot/toolbox/simulink/simulink/+Simulink/parallelMdlRefHostConfigCheckFcn.m`. The `parallelMdlRefHostConfigCheckFcn` function tries to establish a common compiler across the MATLAB Distributed Computing Server client and workers.

For more information about parallel builds, see “Reduce Build Time for Referenced Models” on page 25-45.

Preventing Resource Conflicts (Optional)

Hook files are optional TLC and MATLAB program files that are invoked at well-defined stages of the build process. Hook files let you customize the build process and communicate information between various phases of the process.

If you are adapting your custom target for code generation compatibility with model reference features, consider adding checks to your hook files for handling referenced models differently than top models to prevent resource conflicts.

For example, consider adding the following check to your `STF_make_rtw_hook.m` file:

```
% Check if this is a referenced model
mdlRefTargetType = get_param(codeGenModelName, 'ModelReferenceTargetType');
isNotModelRefTarget = strcmp(mdlRefTargetType, 'NONE'); % NONE, SIM, or RTW
if isNotModelRefTarget
    % code that is specific to the top model
else
    % code that is specific to the referenced model
end
```

You may need to do a similar check in your TLC code.

```
%if !IsModelReferenceTarget()
    %% code that is specific to the top model
%else
    %% code that is specific to the referenced model
%endif
```

More About

- “About Embedded Target Development” on page 37-2
- “Sample Custom Targets” on page 37-9
- “Customize System Target Files” on page 37-29

Support Compiler Optimization Level Control

This section describes how to configure a custom embedded target to support compiler optimization level control. Without the described modifications, you will not be able to use the **Compiler optimization level** parameter on the **All Parameters** tab of the Configuration Parameters dialog box to control the compiler optimization level for building generated code. (For more information about compiler optimization level control, see “Compiler optimization level”.)

In this section...

“About Compiler Optimization Level Control and Custom Targets” on page 37-95

“Declaring Compiler Optimization Level Control Compliance” on page 37-95

“Providing Compiler Optimization Level Control Support in the Target Makefile” on page 37-96

About Compiler Optimization Level Control and Custom Targets

The requirements for supporting compiler optimization level control are as follows:

- The target must be derived from the GRT target or the ERT target.
- The system target file (STF) must declare compiler optimization level control compliance, as described in “Declaring Compiler Optimization Level Control Compliance” on page 37-95.
- The target makefile must honor the setting for **Compiler optimization level**, as described in “Providing Compiler Optimization Level Control Support in the Target Makefile” on page 37-96.

For an example that shows how to configure custom target optional features, see “Customize System Target Files” on page 37-29.

Declaring Compiler Optimization Level Control Compliance

To declare compiler optimization level control compliance for your target, you must implement a callback function that sets the `CompOptLevelCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `CompOptLevelCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'CompOptLevelCompliant','on');  
slConfigUISetEnabled(hDlg,hSrc,'CompOptLevelCompliant',false);
```

For more information about the STF callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

When the `CompOptLevelCompliant` target configuration parameter is set to `on`, the **Compiler optimization level** parameter is displayed in the **Code Generation** pane of the Configuration Parameters dialog box for your model.

Providing Compiler Optimization Level Control Support in the Target Makefile

As part of supporting compiler optimization level control for your target, you must modify the target makefile to honor the setting for **Compiler optimization level**. Use a GRT or ERT target provided by MathWorks as a model for making the modifications.

More About

- “About Embedded Target Development” on page 37-2
- “Sample Custom Targets” on page 37-9
- “Customize System Target Files” on page 37-29

Support C Function Prototype Control

This section describes how to configure a custom embedded target to support C function prototype control. Without the described modifications, you will not be able to use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box to control the function prototypes of initialize and step functions that are generated for your model. (For more information about C function prototype control, see “Control Generation of Function Prototypes” in the Embedded Coder documentation.)

In this section...

“About C Function Prototype Control and Custom Targets” on page 37-97

“Declaring C Function Prototype Control Compliance” on page 37-97

“Providing C Function Prototype Control Support in the Custom Static Main Program” on page 37-98

About C Function Prototype Control and Custom Targets

The requirements for supporting C function prototype control are as follows:

- The target must be derived from the ERT target.
- The system target file (STF) must declare C function prototype control compliance, as described in “Declaring C Function Prototype Control Compliance” on page 37-97.
- If your target uses a custom static main program, and if a nondefault function prototype control configuration is associated with a model, the static main program must call the function prototype controlled initialize and step functions, as described in “Providing C Function Prototype Control Support in the Custom Static Main Program” on page 37-98.

For an example that shows how to configure custom target optional features, see “Customize System Target Files” on page 37-29.

Declaring C Function Prototype Control Compliance

To declare C function prototype control compliance for your target, you must implement a callback function that sets the `ModelStepFunctionPrototypeControlCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered

whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `ModelStepFunctionPrototypeControlCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'ModelStepFunctionPrototypeControlCompliant','on');  
slConfigUISetEnabled(hDlg,hSrc,'ModelStepFunctionPrototypeControlCompliant',false);
```

For more information about the STF callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

When the `ModelStepFunctionPrototypeControlCompliant` target configuration parameter is set to `on`, you can use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box to control the function prototypes of initialize and step functions that are generated for your model.

Providing C Function Prototype Control Support in the Custom Static Main Program

If your target uses a custom static main program, and if a nondefault function prototype control configuration is associated with a model, you must update the static main program to call the function prototype controlled initialize and step functions. You can do this in either of the following ways:

- 1 Manually adapt your static main program to declare model data and call the function prototype controlled initialize and step functions.
- 2 Generate your main program using **Generate an example main program** on the **Templates** pane of the Configuration Parameters dialog box. The generated main program declares model data and calls the function prototype controlled initialize and step function.

More About

- “About Embedded Target Development” on page 37-2

- “Sample Custom Targets” on page 37-9
- “Customize System Target Files” on page 37-29

Support C++ Class Interface Control

This section describes how to configure a custom embedded target to support C++ class interface control. Without the described modifications, you will not be able to use C++ `class` code interface packaging and the **Configure C++ Class Interface** button on the **Interface** pane of the Configuration Parameters dialog box to generate and configure C++ class interfaces to model code. (For more information about C++ class interface control, see “Control Generation of C++ Class Interfaces” in the Embedded Coder documentation.)

In this section...

“About C++ Class Interface Control and Custom Targets” on page 37-100

“Declaring C++ Class Interface Control Compliance” on page 37-100

“Providing C++ Class Interface Control Support in the Custom Static Main Program” on page 37-101

About C++ Class Interface Control and Custom Targets

The requirements for supporting C++ class interface control are as follows:

- The target must be derived from the ERT target.
- The system target file (STF) must declare C++ class interface control compliance, as described in “Declaring C++ Class Interface Control Compliance” on page 37-100.

For an example that shows how to configure custom target optional features, see “Customize System Target Files” on page 37-29.

Declaring C++ Class Interface Control Compliance

To declare C++ class interface control compliance for your target, you must implement a callback function that sets the `CPPClassGenCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `CPPClassGenCompliant` flag as follows:

```
sIConfigUISetVal(hDlg,hSrc,'CPPClassGenCompliant','on');  
sIConfigUISetEnabled(hDlg,hSrc,'CPPClassGenCompliant',false);
```

For more information about the STF callback API, see the `sIConfigUIGetVal`, `sIConfigUISetEnabled`, and `sIConfigUISetVal` function reference pages.

When the `CPPClassGenCompliant` target configuration parameter is set to `on`, you can use the `C++ class` code interface packaging and the **Configure C++ Class Interface** button on the **Interface** pane of the Configuration Parameters dialog box to generate and configure C++ class interfaces to model code.

Providing C++ Class Interface Control Support in the Custom Static Main Program

Selecting `C++ class` code interface packaging for your model turns on the model option **Generate an example main program**. With this option on, code generation generates an example main program, `ert_main.cpp`. The generated example main program declares model data and calls the C++ class interface configured model step method, and illustrates how the generated code can be deployed.

To customize the build process and disable generation and inclusion of an example main program, see the `setTargetProvidesMain` function. Disabling example main generation permits including a custom main program.

More About

- “About Embedded Target Development” on page 37-2
- “Sample Custom Targets” on page 37-9
- “Customize System Target Files” on page 37-29

Support Concurrent Execution of Multiple Tasks

If a custom embedded target must support concurrent execution of multiple tasks on a multicore platform, the target must declare support for concurrent execution by setting the target configuration option `ConcurrentExecutionCompliant`. Otherwise, you will not be able to configure a multicore target model for concurrent execution.

If `ConcurrentExecutionCompliant` is not already configured for your custom target, you can set the option in the following ways:

- Include the following code directly in your system target file (*mytarget.tlc*):

```
rtwgensettings.SelectCallback = 'slConfigUISetVal(hDlg,hSrc,...
''ConcurrentExecutionCompliant'', 'on');';
rtwgensettings.ActivateCallback = 'slConfigUISetVal(hDlg,hSrc,...
''ConcurrentExecutionCompliant'', 'on');';
```

- Implement a callback function that sets the `ConcurrentExecutionCompliant` option, and then install the callback function in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The callback function is triggered whenever the user selects the target in the System Target File Browser. For example, the following STF code installs a `SelectCallback` function named `custom_select_callback_handler`:

```
rtwgensettings.SelectCallback = 'custom_select_callback_handler(hDlg,hSrc)';
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in STF callback functions. They should be passed in without alteration.

Your callback function should set the `ConcurrentExecutionCompliant` option as follows:

```
slConfigUISetVal(hDlg,hSrc, 'ConcurrentExecutionCompliant', 'on');
slConfigUISetEnabled(hDlg,hSrc, 'ConcurrentExecutionCompliant', false);
```

For more information about the STF callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` function reference pages.

When the `ConcurrentExecutionCompliant` target configuration option is set to 'on', you can select the custom target and configure your multicore target model for concurrent execution.

For an example that shows how to configure custom target optional features, see “Customize System Target Files” on page 37-29.

More About

- “About Embedded Target Development” on page 37-2
- “Sample Custom Targets” on page 37-9
- “Customize System Target Files” on page 37-29

Interface to Development Tools

Unless you are developing a target purely for code generation purposes, you will want your embedded target to support a complete build process. A full post-code generation build process includes

- Compilation of generated code
- Linking of compiled code and runtime libraries into an executable program module (or some intermediate representation of the executable code, such as S-Rec format)
- Downloading the executable to target hardware with a debugger or other utility
- Initiating execution of the downloaded program

Supporting a complete build process is inherently a complex task, because it involves interfacing to cross-development tools and utilities that are external to the code generator.

If your development tools can be controlled with traditional makefiles and a make utility such as `gmake`, it may be relatively simple for you to adapt existing target files (such as the `ert.tlc` and `ert.tmf` files provided by the Embedded Coder software) to your requirements. This approach is discussed in “Template Makefile Approach” on page 37-105.

In this section...
“About Interfacing to Development Tools” on page 37-104
“Template Makefile Approach” on page 37-105
“Interface to an Integrated Development Environment” on page 37-105

About Interfacing to Development Tools

Automating your build process through a modern integrated development environment (IDE) presents a different set of challenges. Each IDE has its own way of representing the set of source files and libraries for a project and for specifying build arguments. Interfacing to an IDE may require generation of specialized file formats required by the IDE (for example, project files) and, and also may require the use of inter-application communication (IAC) techniques to run the IDE. One such approach to build automation is discussed in “Interface to an Integrated Development Environment” on page 37-105.

Template Makefile Approach

A template makefile provides information about your model and your development system. The build process uses this information to create a makefile (.mk file) to build an executable program. The code generator provides a number of template makefiles suitable for development computer compilers, such as LCC (`ert_lcc.tmf`) and Microsoft Visual C++ (`ert_vc.tmf`).

Adapting one of the existing template makefiles to your cross-compiler's make utility may require little more than copying and renaming the template makefile in accordance with the conventions of your project.

If you need to make more extensive modifications, you need to understand template makefiles in detail. For a detailed description of the structure of template makefiles and of the tokens used in template makefiles, see “Customize Template Makefiles” on page 37-62.

The following topics supplement the basic template makefile information:

- “Supporting Multiple Development Environments” on page 37-48
- “Supplying Development Environment Information to Your Template Makefile” on page 37-26

Interface to an Integrated Development Environment

This section describes techniques that have been used to integrate embedded targets with integrated development environment (IDEs), including

- How to generate a header file containing directives to define variables (and their values) required by a non-makefile based build.
- Some problems and solutions specific to interfacing embedded targets with the Freescale Semiconductor CodeWarrior IDE. The examples provided should help you to deal with similar interfacing problems with your particular IDE.
- “Generating a CPP_REQ_DEFINES Header File” on page 37-105
- “Interfacing to the Freescale CodeWarrior IDE” on page 37-107

Generating a CPP_REQ_DEFINES Header File

In template makefiles, the token `CPP_REQ_DEFINES` is expanded and replaced with a list of parameter settings entered with various dialog boxes. This variable often contains

information such as MODEL (name of generating model), NUMST (number of sample times in the model), MT (model is multitasking or not), and numerous other parameters (see “Template Makefiles and Tokens” on page 37-62).

The makefile mechanism handles the CPP_REQ_DEFINES token automatically. If your target requires use of a project file, rather than the traditional makefile approach, you can generate a header file containing directives to define these variables and provide their values.

The following TLC file, `gen_rtw_req_defines.tlc`, provides an example. The code generates a C header file, `cpp_req_defines.h`. The information required to generate each `#define` directive is derived either from information in the `model.rtw` file (e.g., `CompiledModel.NumSynchronousSampleTimes`), or from make variables from the `rtwoptions` structure (for example, `PurelyIntegerCode`).

```
%% File: gen_rtw_req_defines_h.tlc
%openfile CPP_DEFINES = "cpp_req_defines.h"
#ifdef _CPP_REQ_DEFINES_
#define _CPP_REQ_DEFINES_
#define MODEL %<CompiledModel.Name>
#define ERT 1
#define NUMST %<CompiledModel.NumSynchronousSampleTimes>
#define TID01EQ %<CompiledModel.FixedStepOpts.TID01EQ>
%%
%if CompiledModel.FixedStepOpts.SolverMode == "MultiTasking"
#define MT 1
#define MULTITASKING 1
%else
#define MT 0
#define MULTITASKING 0
#endif
%%
#define MAT_FILE 0
#define INTEGER_CODE %<PurelyIntegerCode>
#define ONESTEPFCN %<CombineOutputUpdateFcns>
#define TERMFCN %<IncludeMdlTerminateFcn>
%%
#define MULTI_INSTANCE_CODE 0
#define HAVESTDIO 0
#endif
%closefile CPP_DEFINES
```


Interfacing to the Freescale CodeWarrior IDE

Interfacing an embedded target's build process to the CodeWarrior IDE requires that two problems must be dealt with:

- The build process must generate a CodeWarrior compatible project file. This problem, and a solution, is discussed in “XML Project Import” on page 37-107. The solution described is applicable to ASCII project file formats.
- During code generation, the target must automate a CodeWarrior session that opens a project file and builds an executable. This task is described in “Build Process Automation” on page 37-112. The solution described is applicable to IDEs that can be controlled with Microsoft Component Object Model (COM) automation.

XML Project Import

This section illustrates how to use the Target Language Compiler (TLC) to generate an eXtensible Markup Language (XML) file, suitable for import into the CodeWarrior IDE, that contains information about the source code generated by an embedded target.

The choice of XML format is dictated by the fact that the CodeWarrior IDE supports project export and import with XML files. As of this writing, native CodeWarrior project files are in a proprietary binary format.

Note that if your target needs to support some other compiler's project file format, you can apply the techniques shown here to other ASCII file formats (see “Generating a CPP_REQ_DEFINES Header File” on page 37-105).

To illustrate the basic concept, consider a hypothetical XML file exported from a CodeWarrior stationery project. The following is a partial listing:

```
<target>
  <settings>
    ...
  <\settings>
  <file><name>foo.c<\name>
  <\file>
  ...
  <file><name>foobar.c<\name>
  <\file>
  <fileref><name>foo.c<\name>
  <\fileref>
  ...
  <fileref><name>foobar.c<\name>
```

```
<\fileref>  
<\target>
```

Insert this XML code into an `%openfile/%closefile` block within a TLC file, `test.tlc`, as shown below.

```

%% test.tlc
%% This code will generate a file model_project.xml,
%% where model is the generating model name specified in
%% the CompiledModel.Name field of the model.rtw file.
%openfile XMLFileContents = %<CompiledModel.Name>_project.xml
<target>
  <settings>
    ...
  <\settings>
  <file><name>%<CompiledModel.Name>.c<\name>
  <\file>
  ...
  <file><name>foobar.c<\name>
  <\file>
  <fileref><name>%<CompiledModel.Name>.c<\name>
  <\fileref>
  ...
  <fileref><name>foobar.c<\name>
  <\fileref>
<\target>
%closefile XMLFileContents
%selectfile NULL_FILE

```

Note the use of the TLC token `CompiledModel.Name`. The token is resolved and the resulting filename is included in the output stream. You can specify other information, such as paths and libraries, in the output stream by specifying other tokens defined in `model.rtw`. For example, `System.Name` may be defined as `<Root>/Subsystem1`.

Now suppose that `test.tlc` is invoked during a target's build process, where the generating model is `mymodel`. This should be done after the `codegenentry` statement. For example, `test.tlc` could be included directly in the system target file:

```

#include "codegenentry.tlc"
#include "test.tlc"

```

Alternatively, the `%include "test.tlc"` directive could be inserted into the `mytarget_genfiles.tlc` hook file, if present.

TLC tokens such as

```

<file><name>%<CompiledModel.Name>.c<\name>

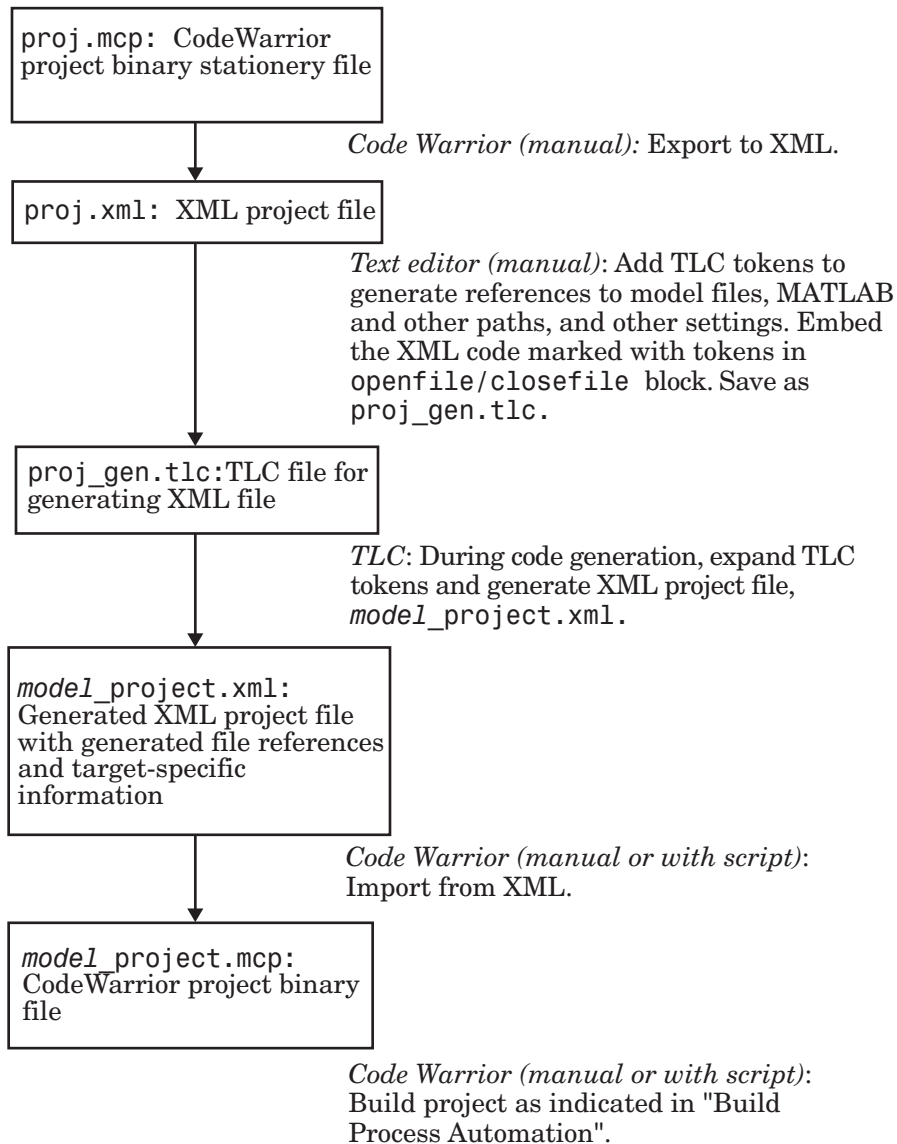
```

are expanded, with the `CompiledModel` record in the `mymodel.rtw` file, as in

```
<file><name>mymodel.c<\name>
```

`test.tlc` generates an XML file, file `model_project.xml`, from a model. `model_project.xml` contains references to generated code files. `model_project.xml` can be imported into the CodeWarrior IDE as a project.

The following flowchart summarizes this process.



Note This process has drawbacks. First, manually editing an XML file exported from a CodeWarrior stationery project can be a laborious task, involving modification of a few

dozen lines embedded within several thousand lines of XML code. Second, if you make changes to the CodeWarrior project after importing the generated XML file, the XML file must be exported and manually edited once again.

Build Process Automation

An application that supports COM automation can control other applications that include a COM interface. Using MATLAB COM automation functions, a MATLAB file can command a COM-compatible development system to execute tasks required by the build process.

The MATLAB COM automation functions described in this section are documented in “Call COM Objects”.

For information about automation commands supported by the CodeWarrior IDE, see your CodeWarrior documentation.

COM automation is used by some embedded targets to automate the CodeWarrior IDE to execute tasks such as:

- Opening a new CodeWarrior session
- Configure a project
- Loading a CodeWarrior project file
- Removing object code from the project
- Building or rebuilding the project
- Debug an application

COM technology automates certain repetitive tasks and allows the user to interact directly with the external application. For example, when the end user of the embedded targets capability initiates a build, the target quickly invokes CodeWarrior actions and leaves a project built and ready to run with the IDE.

Example COM Automation Functions

The functions below use the MATLAB `actxserver` command to invoke COM functions for controlling the CodeWarrior IDE from a MATLAB file:

- `CreateCWComObject`: Create a COM connection to the CodeWarrior IDE.
- `OpenCW`: Open the CodeWarrior IDE without opening a project.

- **OpenMCP**: Open the CodeWarrior project file (.mcp file) specified by the input argument.
- **BuildCW**: Open the specified .mcp file, remove object code, and build project.

These functions are examples; they do not constitute a full implementation of a COM automation interface. If your target creates the project file during code generation, the top-level **BuildCW** function should be called after the code generation process is completed. Normally **BuildCW** would be called from the **exit** method of your *STF_make_rtw_hook.m* file (see “STF_make_rtw_hook.m” on page 37-21).

In the code examples, the variable **in_qualifiedMCP** is assumed to store a fully qualified path to a CodeWarrior project file (for example, path, filename, and extension). For example:

```
in_qualifiedMCP = 'd:\work\myproject.mcp';
```

In actual practice, your code is responsible for determining the conventions used for the project filename and location. One simple convention would be to default to a project file *model.mcp*, located in your target's build folder.

```
%=====
% Function: CreateCWComObject
% Abstract: Creates the COM connection to CodeWarrior
%
function ICodeWarriorApp = CreateCWComObject
    vprint([mfilename ' : creating CW com object']);
    try
        ICodeWarriorApp = actxserver('CodeWarrior.CodeWarriorApp');
    catch
        error(['Error creating COM connection to ' ComObj ...
            '. Verify that CodeWarrior is installed. Verify COM access to
CodeWarrior outside of MATLAB.']);
    end
    return;

%=====
% Function: OpenCW
% Abstract: Opens CodeWarrior without opening a project. Returns the
           handle ICodeWarriorApp.
%
function ICodeWarriorApp = OpenCW()
    ICodeWarriorApp = CreateCWComObject;
    CloseAll;
    OpenMCP(in_qualifiedMCP);

%=====
% Function: OpenMCP
```

```

% Abstract: open an MCP project file
%
function OpenMCP(in_qualifiedMCP)
% Argument checking. This method requires valid project file.
if ~exist(in_qualifiedMCP)
    error(['filename ': Missing or empty project file argument']);
end
if isempty(in_qualifiedMCP)
    error(['filename ': Missing or empty project file argument']);
end
ICodeWarriorApp = CreateCWComObject;
vprint(['filename ': Importing]);
try
    ICodeWarriorProject = ...
        invoke(ICodeWarriorApp.Application,...
            'OpenProject', in_qualifiedMCP,...
            1,0,0);
catch
    error(['Error using COM connection to import project. ' ...
        'Verify that CodeWarrior is installed. Verify COM access to
CodeWarrior outside of MATLAB.']);
end

%=====
% Function: BuildCW
% Abstract: Opens CodeWarrior.
%           Opens the specified CodeWarrior project.
%           Deletes objects.
%           Builds.
%
function ICodeWarriorApp = BuildCW(in_qualifiedMCP)
% ICodeWarriorApp = BuildCW;
ICodeWarriorApp = CreateCWComObject;
CloseAll;
OpenMCP(in_qualifiedMCP);
try
    invoke(ICodeWarriorApp.DefaultProject,'RemoveObjectCode', 0, 1);
catch
    error(['Error using COM connection to remove objects of current project. ' ...
        'Verify that CodeWarrior is installed. Verify COM access to
CodeWarrior outside of MATLAB.']);
end
try
    invoke(ICodeWarriorApp.DefaultProject,'BuildAndWaitToComplete');
catch
    error(['Error using COM connection to build current project. ' ...
        'Verify that CodeWarrior is installed. Verify COM access to
CodeWarrior outside of MATLAB.']);
end
end

```

More About

- “About Embedded Target Development” on page 37-2

- “Sample Custom Targets” on page 37-9
- “Customize System Target Files” on page 37-29

Device Drivers

Device drivers that communicate with target hardware are essential to many real-time development projects.

You can integrate existing C (or C++) device driver functions into Simulink models by using the Legacy Code Tool. When you use the code generator to generate code from a model, the Legacy Code Tool can insert a call to your C function into the generated code. For details, see “Build S-Function With Legacy Code Tool” on page 24-27 and “Integrate C Functions into Simulink Models with Legacy Code Tool”.

More About

- “About Embedded Target Development” on page 37-2
- “Sample Custom Targets” on page 37-9
- “Customize System Target Files” on page 37-29