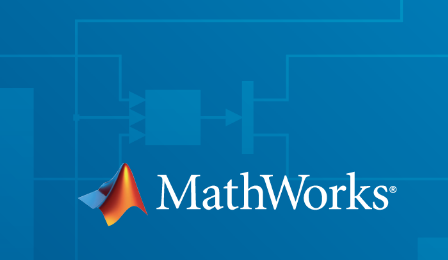
Simulink® Coder™

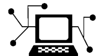
User's Guide

R2014b

MATLAB® & SIMULINK®



How to Contact MathWorks



Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Simulink® Coder™ User's Guide

© COPYRIGHT 2011–2014 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

April 2011	Online only	New for Version 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)

Check Bug Reports for Issues and Fixes

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

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

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

Contents

Model Architecture and Design

1 Modeling

Configure a Model for Code Generation **1-2**

Scheduling **1-4**
About Scheduling **1-4**
Single-Tasking and Multitasking Execution Modes **1-4**
Handle Rate Transitions **1-13**
Single-Tasking and Multitasking Model Execution . . . **1-26**
Handle Asynchronous Events **1-31**
Timers **1-69**
Configure Scheduling **1-79**

Supported Products and Block Usage **1-81**
Related Products **1-81**
Simulink Built-In Blocks That Support Code Generation **1-83**
Simulink Block Data Type Support Table **1-101**
Block Set Support for Code Generation **1-101**

Modeling Semantic Considerations **1-102**
Data Propagation **1-102**
Sample Time Propagation **1-104**
Latches for Subsystem Blocks **1-105**
Block Execution Order **1-105**
Algebraic Loops **1-106**

2 Subsystems

Code Generation of Subsystems **2-2**
Subsystem Code Dependence **2-3**

Generate Code and Executables for Individual Subsystem **2-4**
Subsystem Build Limitations **2-6**

Inline Subsystem Code **2-7**
Configure Subsystem to Inline Code **2-7**
Exceptions to Inlining **2-8**

Generate Subsystem Code as Separate Function and Files **2-10**

Generate Reusable Function for Identical Subsystems Within a Model **2-11**
Considerations for Function Packaging Options Auto and Reusable function **2-13**

Optimize Code for Identical Nested Subsystems **2-14**

Generate Reusable Code for Subsystems Containing S-Function Blocks **2-15**

Generate Reusable Code from Stateflow Charts **2-16**

Code Reuse Limitations for Subsystems **2-17**
Blocks That Prevent Code Reuse **2-17**
Code Reuse Limitations for Subsystems Shared Across Referenced Models **2-18**

Code Reuse For Subsystems Shared Across Models . . **2-19**

Reusable Library Subsystem **2-20**
Code Generation of a Reusable Library Subsystem . . . **2-20**
Reusable Library Subsystem Code Placement and Naming **2-21**
Reusable Library Subsystem in the Top Model **2-21**

Reusable Library Subsystem Connected to Root Outport **2-21**

Code Generation of Constant Parameters **2-22**

Shared Constant Parameters for Code Reuse **2-23**
Suppress Shared Constants in the Generated Code . . . **2-24**
Shared Constant Parameters Limitations **2-26**

Generate Reusable Code for Subsystems Shared Across Models **2-27**

Determine Why Subsystem Code Is Not Reused **2-35**
Review Subsystems Section of HTML Code Generation Report **2-35**
Compare Subsystem Checksum Data **2-35**

3 Code Generation of Functions and Function Callers

Modeling Functions and Callers for Code Generation . **3-2**
Functions and Callers **3-2**
Input and Output Arguments **3-2**
Function and Function Caller Definitions Across Models **3-3**
Code Generation Files **3-3**

Generate Code for Functions and Callers **3-6**
Generate Code for the Function Definition **3-6**
Generate Code for the Caller Definition **3-8**

4 Referenced Models

Code Generation for Referenced Models **4-2**

Generate Code for Referenced Models **4-4**
About Generating Code for Referenced Models **4-4**
Create and Configure the Subsystem **4-4**
Convert Model to Use Model Referencing **4-7**
Generate Model Reference Code for a GRT Target **4-11**
Work with Code Generation Folders **4-14**

Code Generation Folder Structure for Model Reference Targets **4-15**

Configure Referenced Models **4-16**

Build Model Reference Targets **4-17**
Reduce Change Checking Time **4-17**

Simulink Coder Model Referencing Requirements . . . **4-18**
Configuration Parameter Requirements **4-18**
Naming Requirements **4-21**
Custom Target Requirements **4-22**

Storage Classes for Signals Used with Model Blocks . . **4-23**
Storage Classes for Parameters Used with Model Blocks **4-23**
Signal Name Mismatches Across Model Reference Boundary **4-24**

Inherited Sample Time for Referenced Models **4-26**

Customize Library File Suffix and File Type **4-28**

Reusable Code and Referenced Models **4-29**
General Considerations **4-29**
Code Reuse and Model Blocks with Root Inport or Outport Blocks **4-29**

Simulink Coder Model Referencing Limitations **4-33**
Customization Limitations **4-33**
Data Logging Limitations **4-33**
State Initialization Limitation **4-34**
Reusability Limitations **4-34**
S-Function Limitations **4-35**
Simulink Tool Limitations **4-35**
Subsystem Limitations **4-35**

Target Limitations 4-35
Other Limitations 4-36

5 Combined Models

Combined Models **5-2**

Use GRT with Reusable Function Packaging to Combine Models **5-3**
Share Data Across Models **5-3**
Timing Issues **5-3**
Data Logging and External Mode Support **5-4**

6 Configure Model Parameters

Platform Options for Development and Deployment . . **6-2**

Configure Test and Production Target Hardware **6-3**
Identify the Device Vendor **6-4**
Identify the Device Type **6-5**
Register Additional Device Vendor and Device Type Values **6-5**
Set Bit Lengths for Device Data Types **6-8**
Set Byte Ordering Used By Device **6-10**
Set Quotient Rounding Behavior for Signed Integer Division **6-10**
Set Arithmetic Right Shift Behavior for Signed Integers **6-11**
Update Release 14 Hardware Configuration **6-12**

Configure Production Hardware Characteristics **6-13**

Configure Test Hardware Characteristics **6-14**

Control the Location for Generated Files **6-17**

Control Generated Files Location Used for Simulation 6-19

Control the Location for Code Generation Files 6-21

Override Build Folder Settings for Current Session . . 6-23

7 Model Protection

Protect a Referenced Model 7-2
Requirements for Protecting a Model 7-3

Harness Model 7-4

Protected Model Report 7-5

Code Generation Support in a Protected Model 7-6
Protected Model Requirements to Support Code Generation 7-6

Protected Model File 7-7

Create a Protected Model 7-9

Protected Model Creation Settings 7-14
Open Read-Only View of Model 7-15
Simulate 7-15
Generate Code 7-15

Test the Protected Model 7-16

Save Base Workspace Definitions 7-18

Package a Protected Model 7-19

Specify Custom Obfuscator for Protected Model 7-20

Data, Function, and File Definition

8 Data Representation

Enumerations **8-2**
About Enumerated Data Types **8-2**
Default Code for an Enumerated Data Type **8-2**
Specify Enumerated Data Type **8-3**
Type Casting for Enumerations **8-4**
Override Default Methods (Optional) **8-5**
Enumerated Type Limitations **8-8**

Structure Parameters and Generated Code **8-9**
About Structure Parameters and Generated Code **8-9**
Configure Structure Parameters for Generated Code . . . **8-9**
Control Name of Structure Parameter Type **8-10**

Parameters **8-11**
About Parameters **8-11**
Nontunable Parameter Storage **8-12**
Tunable Parameter Storage **8-14**
Tunable Parameter Storage Classes **8-15**
Declare Tunable Parameters **8-17**
Tunable Expressions **8-21**
Linear Block Parameter Tunability **8-25**
Configuration Parameter Quick Reference Diagram . . **8-26**
Generated Code for Parameter Data Types **8-27**
Tunable Workspace Parameter Data Type Considerations **8-32**
Tune Parameters **8-34**
Parameter Objects **8-35**

Signals **8-46**
About Signals **8-46**
Signal Storage Concepts **8-47**
Signals with Auto Storage Class **8-49**
Signals with Test Points **8-53**
Interface Signals to External Code **8-53**
Symbolic Naming Conventions for Signals **8-55**

Summary of Signal Storage Class Options 8-56
Interfaces for Monitoring Signals 8-57
Signal Objects 8-57
Initialize Signals and States Using Signal Objects 8-65

States **8-74**
About States 8-74
Symbolic Names for Continuous Block States 8-74
State Attributes 8-76
State Storage 8-76
State Storage Classes 8-77
Interface States to External Code 8-78
Symbolic Names for States 8-80
Control Code Generation for Block States 8-83
Summary of State Storage Class Options 8-83

Data Stores **8-85**
About Data Stores 8-85
Storage Classes for Data Store Memory Blocks 8-85
Generate Code for Data Store Memory Blocks 8-87
Nonscalar Data Stores in Generated Code 8-88
Data Store Buffering in Generated Code 8-90

9 Entry-Point Functions and Scheduling

Entry-Point Functions and Scheduling **9-2**

Generate Reentrant Code from Top-Level Models **9-4**

Generate C++ Class Interface to Model or Subsystem Code **9-5**
About C++ Class Code Interface Packaging 9-5
Generate C++ Class Interface to Model Code 9-5
Generate C++ Class Interface to Nonvirtual Subsystem Code 9-7
C++ Class Interface Limitations 9-7

About Model Execution **9-9**

Non-Real-Time Single-Tasking Systems 9-11

Non-Real-Time Multitasking Systems 9-12

Real-Time Single-Tasking Systems 9-14

Real-Time Multitasking Systems 9-16

Multitasking Systems Using Real-Time Tasking Primitives 9-18

Program Timing 9-20

Program Execution 9-22

External Mode Communication 9-23

Data Logging in Single-Tasking and Multitasking Model Execution 9-24

Rapid Prototyping and Embedded Model Execution Differences 9-25

Rapid Prototyping Model Functions 9-26

Code Generation

10 Configuration

Code Generation Configuration 10-2

Open the Model Configuration for Code Generation . . 10-3

Configure a Model Programmatically 10-4

Checking Model and Configuration with Model Advisor 10-6

Check Model for Code Efficiency 10-7

Application Objectives 10-8
High-Level Code Generation Objectives 10-8
Check and Configure Model for Code Generation Objectives 10-9
Check and Configure Model for Code Generation Objectives Using Configuration Parameters Dialog Box 10-11

Target 10-12
Hardware 10-12
Available Targets 10-12
About Targets and Code Formats 10-15
Types of Target Code Formats 10-16
Targets and Code Formats 10-27
Targets and Code Styles 10-28
Backwards Compatibility of Code Formats 10-29
Select a Target 10-32
Template Makefiles and Make Options 10-35
Custom Targets 10-41
Standard Math Libraries 10-41
Change the Standard Math Library 10-41
Specify Target Interfaces 10-41

Change Programming Language 10-46

Configure Code Comments 10-47

Construction of Generated Identifiers 10-48

Identifier Name Collisions and Mangling 10-49
Identifier Name Collisions with Referenced Models . . 10-49

Specify Identifier Length to Avoid Naming Collisions 10-50

Specify Reserved Names for Generated Identifiers . . 10-51

Reserved Keywords 10-52
C Reserved Keywords 10-52

C++ Reserved Keywords 10-53
Reserved Keywords for Code Generation 10-53
Simulink Coder Code Replacement Library Keywords 10-54

Debug 10-56

11 Source Code Generation

Initiate Code Generation 11-2

Model and Test Environment 11-3
About This Example 11-3
Functional Design of the Model 11-4
View the Top Model 11-4
View the Subsystems 11-5
Simulation Test Environment 11-6
Run Simulation Tests 11-11
Key Points 11-12
Learn More 11-13

Configure Model and Generate Code 11-14
About This Example 11-14
Configure the Model for Code Generation 11-2
Save Your Model Configuration as a MATLAB Function 11-16
Check the Model for Adverse Conditions and Code Generation Settings 11-17
Generate Code for the Model 11-17
Review the Generated Code 11-17
Generate an Executable 11-18
Key Points 11-19
Learn More 11-19

Configure Data Interface 11-20
About This Example 11-20
Declare Data 11-20
Use Data Objects 11-21
Add New Data Objects 11-25
Enable Data Objects for Generated Code 11-25

Effects of Simulation on Data Typing 11-26
Manage Data 11-27
Key Points 11-28
Learn More 11-28

Call External C Functions 11-29
About This Example 11-29
Include External C Functions in a Model 11-30
Create a Block That Calls a C Function 11-30
Validate External Code in the Simulink Environment 11-32
Validate C Code as Part of a Model 11-33
Call a C Function from Generated Code 11-35
Key Points 11-35
Learn More 11-35

Reload Generated Code 11-36

Generated Source Files and File Dependencies 11-37
About Generated Files and File Dependencies 11-37
Header Dependencies When Interfacing Legacy/Custom Code with Generated Code 11-39
Dependencies of the Generated Code 11-48
Specify Include Paths in Simulink Coder Generated Source Files 11-53

Files and Folders Created by Build Process 11-56
Files Created During the Build Process 11-56
Folders Used During the Build Process 11-61

How Code Is Generated From a Model 11-63
Model Compilation 11-63
Code Generation 11-63

Code Generation of Matrices and Arrays 11-65
Simulink Coder Matrix Parameters 11-66
Internal Data Storage for Complex Number Arrays . . 11-68

Generated Code Considerations 11-69
Requirements for Signed Integer Representation 11-69
Subnormal Numbers 11-69
Specify Order of `rtwtypes.h` Include File 11-70
Default Folder for Code Generation 11-70

About Shared Utility Code **11-71**

Controlling Shared Utility Code Placement **11-72**

rtwtypes.h and Shared Utility Code **11-73**

Incremental Shared Utility Code Generation and Compilation **11-74**

Shared Utility Checksum **11-75**

Shared Fixed-Point Utility Functions **11-77**

Share User-Defined Data Types Across Models **11-79**
About Sharing Data Types **11-79**
Example: Sharing Simulink Data Type Objects **11-80**
Example: Sharing Enumerated Data Types **11-81**

Generating Code Using Simulink® Coder™ **11-84**

12 Report Generation

Reports for Code Generation **12-2**

HTML Code Generation Report Location **12-3**

HTML Code Generation Report for Referenced Models **12-4**

Generate a Code Generation Report **12-5**

Generate Code Generation Report After Build Process **12-6**

Open Code Generation Report **12-8**
Limitation **12-8**

Generate Code Generation Report Programmatically **12-10**

Search Code Generation Report **12-11**

View Code Generation Report in Model Explorer . . . **12-12**

Package and Share the Code Generation Report **12-14**
Package the Code Generation Report **12-14**
View the Code Generation Report **12-15**

Document Generated Code with Simulink Report Generator **12-16**
Generate Code for the Model **12-17**
Open the Report Generator **12-17**
Set Report Name, Location, and Format **12-19**
Include Models and Subsystems in a Report **12-20**
Customize the Report **12-21**
Generate the Report **12-22**

13 Code Replacement for Simulink Models

What Is Code Replacement? **13-2**

Code Replacement Libraries **13-4**

Code Replacement Terminology **13-6**

Code Replacement Limitations **13-9**

Replace Code Generated from Simulink Models **13-10**

Choose a Code Replacement Library **13-13**
About Choosing a Code Replacement Library **13-13**
Explore Available Code Replacement Libraries **13-13**
Explore Code Replacement Library Contents **13-13**

14 Desktops

Rapid Simulations **14-2**
About Rapid Simulation **14-2**
Rapid Simulation Advantage **14-3**
General Rapid Simulation Workflow **14-3**
Identify Rapid Simulation Requirements **14-4**
Configure Inports to Provide Simulation Source Data . **14-6**
Configure and Build Model for Rapid Simulation **14-6**
Set Up Rapid Simulation Input Data **14-8**
Scripts for Batch and Monte Carlo Simulations **14-18**
Run Rapid Simulations **14-18**
Rapid Simulation Target Limitations **14-30**

Generated S-Function Block **14-31**
About Object Libraries **14-31**
Create S-Function Blocks from a Subsystem **14-34**
Tunable Parameters in Generated S-Functions **14-38**
System Target File and Template Makefiles **14-40**
Checksums and the S-Function Target **14-41**
S-Function Target Limitations **14-42**

15 Real-Time Systems

Real-Time System Rapid Prototyping **15-2**
About Real-Time Rapid Prototyping **15-2**
Goals of Real-Time Rapid Prototyping **15-2**
Refine Code With Real-Time Rapid Prototyping **15-3**

Hardware-In-the-Loop (HIL) Simulation **15-5**
About Hardware-In-the-Loop Simulation **15-5**
Set Up and Run HIL Simulations **15-6**

16 External Code Integration

Integration Options **16-2**
About Integration Options **16-2**
Types of External Code Integration **16-2**

Reuse Algorithmic Components in Generated Code . . **16-5**
Reusable Algorithmic Components **16-5**
Integrate External MATLAB Code **16-5**
Integrate External C or C++ Code **16-7**
Integrate Fortran Code **16-10**
Integration Considerations for Reusable Algorithmic Components **16-10**

Deploy Algorithm Code Within a Target Environment **16-12**

Export Generated Algorithm Code for Embedded Applications **16-16**

Export Algorithm Executables for System Simulation **16-19**

Modify External Code for Language Compatibility . . **16-20**

Automate S-Function Generation **16-21**

Integrate External Code Using Legacy Code Tool . . . **16-25**
Legacy Code Tool and Code Generation **16-25**
Generate Inlined S-Function Files for Code Generation **16-26**
Apply Code Style Settings to Legacy Functions **16-26**
Address Dependencies on Files in Different Locations **16-27**
Deploy S-Functions for Simulation and Code Generation **16-28**

Configure Model for External Code Integration **16-29**

Insert Custom Code Blocks **16-32**
Custom Code Library **16-32**
Embed Custom Code Directly Into MdlStart Function **16-35**
Custom Code in Subsystems **16-38**

Preserve User Files in Build Folder 16-39

Insert S-Function Code 16-40
About S-Functions and Code Generation 16-40
Write Noninlined S-Functions 16-45
Write Wrapper S-Functions 16-47
Write Fully Inlined S-Functions 16-56
Write Fully Inlined S-Functions with mdlRTW Routine 16-56
Guidelines for Writing Inlined S-Functions 16-75
Write S-Functions That Support Expression Folding . 16-76
S-Functions That Specify Port Scope and Reusability 16-87
S-Functions That Specify Sample Time Inheritance Rules 16-91
S-Functions That Support Code Reuse 16-93
S-Functions for Multirate Multitasking Environments 16-94
Build Support for S-Functions 16-100

17 Program Building, Interaction, and Debugging

Compiler or IDE Selection and Configuration 17-2
Compilers and the Build Process 17-2
Language Standards Compliance 17-3
Language Considerations 17-3
C++ Language Limitations 17-4
International Character Support 17-5
Choose and Configure Compiler on Microsoft Windows 17-7
Choose and Configure Compiler on UNIX 17-7
Include S-Function Source Code 17-8
Troubleshoot Compiler Configurations 17-8

Program Builds 17-11
Configure the Build Process 17-11
Initiate the Build Process 17-18
Build a Generic Real-Time Program 17-19
Rebuild a Model 17-28
Control Regeneration of Top Model Code 17-29
Reduce Build Time for Referenced Models 17-31
Relocate Code to Another Development Environment 17-35

How Executable Programs Are Built From Models . . 17-40

Build and Run a Program 17-44

Profile Code Performance 17-46
About Profiling Code Performance 17-46
How to Profile Code Performance 17-46
Run Profiling Hooks for Generated Code 17-49
Profiling Limitation 17-49

Data Exchange 17-50
Host/Target Communication 17-50
Logging 17-99
Parameter Tuning 17-110
Data Interchange Using the C API 17-125
ASAP2 Data Measurement and Calibration 17-158
Direct Memory Access to Generated Code 17-171

Performance

18 Optimizations for Generated Code

Optimization Parameters 18-2

Advice About Optimizing Models for Code Generation 18-4

Control Compiler Optimizations 18-5

Optimization Tools and Techniques 18-6

Control Memory Allocation for Time Counters 18-8

Optimization Parameter Dependencies 18-9

Execution Profiling for Generated Code 18-11

19 Defensive Programming

Optimize Code for Floating-Point to Integer Conversions **19-2**
Remove Code That Wraps Out-of-Range Values **19-2**
Remove Code That Maps NaN to Integer Zero **19-2**

Disable Nonfinite Checks or Inlining for Math Functions **19-4**

20 Data Copy Reduction

Optimize Buffers in the Generated Code **20-2**
Configure Buffer Optimizations **20-2**
Example Model **20-2**
Generate Code Without Buffer Optimization **20-3**
Enable Buffer Optimization **20-5**

Minimize Computations and Storage for Intermediate Results **20-8**
About Expression Folding **20-8**
Expression Folding Example **20-9**
Enable Expression Folding **20-11**

Declare Signals as Local Function Data **20-13**

Inline Invariant Signals **20-14**

21 Execution Speed

Inline Parameters **21-2**
Referenced Models **21-2**

Configure Loop Unrolling Threshold **21-4**

Optimize Code Generated for Vector Assignments . . . **21-6**
Configure Model to Optimize Code Generated for Vector Assignments **21-6**
Optimize Code Generated for Vector Assignments Using memcpy **21-7**

Generate Target Optimizations Within Algorithm Code **21-10**

22 Memory Usage

Minimize Memory Requirements During Code Generation **22-2**

Implement Logic Signals as Boolean Data **22-3**

Reduce Memory Requirements for Signals **22-4**

Reuse Memory Allocated for Signals **22-5**

Customize Stack Space Allocation **22-6**

Verification

23 Simulation and Code Comparison

Configure Signal Data for Logging **23-2**

Log Simulation Data **23-4**

Run Executable and Load Data 23-6

Visualize and Compare Results 23-7

Customization

24 Build Process Integration

Control Build Process Compiling and Linking **24-2**

Cross-Compile Code Generated on Microsoft Windows **24-4**

Control Library Location and Naming During Build . **24-7**
Library Control Parameters **24-7**
Specify the Location of Precompiled Libraries **24-9**
Control the Location of Model Reference Libraries . . . **24-10**
Control the Suffix Applied to Library File Names . . . **24-11**

Recompile Precompiled Libraries **24-13**

Customize Post-Code-Generation Build Processing . **24-14**
Build Information Object **24-15**
Program a Post Code Generation Command **24-15**
Define a Post Code Generation Command **24-16**
Suppress Makefile Generation **24-17**

Configure Generated Code with TLC **24-19**
About Configuring Generated Code with TLC **24-19**
Assigning Target Language Compiler Variables **24-19**
Set Target Language Compiler Options **24-20**

Customize Build Process with STF_make_rtw_hook File **24-21**
The STF_make_rtw_hook File **24-21**
Conventions for Using the STF_make_rtw_hook File . **24-21**

STF_make_rtw_hook.m Function Prototype and Arguments **24-22**
Applications for STF_make_rtw_hook.m **24-24**
Control Code Regeneration Using STF_make_rtw_hook.m **24-25**
Use STF_make_rtw_hook.m for Your Build Procedure **24-26**

Customize Build Process with sl_customization.m . . **24-27**
The sl_customization.m File **24-27**
Register Build Process Hook Functions Using sl_customization.m **24-29**
Variables Available for sl_customization.m Hook Functions **24-29**
Example Build Process Customization Using sl_customization.m **24-30**

Replace the STF_rtw_info_hook Mechanism **24-32**

Customize Build to Use Shared Utility Code **24-33**
Modify Template Makefiles to Support Shared Utilities **24-33**

25 Run-Time Data Interface Extensions

Customize an ASAP2 File **25-2**
About ASAP2 File Customization **25-2**
ASAP2 File Structure on the MATLAB Path **25-2**
Customize the Contents of the ASAP2 File **25-3**
ASAP2 Templates **25-4**
Customize Computation Method Names **25-6**
Suppress Computation Methods for FIX_AXIS **25-7**

Create a Transport Layer for External Communication **25-8**
About Creating a Transport Layer for External Communication **25-8**
Design of External Mode **25-8**
External Mode Communications Overview **25-11**
External Mode Source Files **25-12**

Implement a Custom Transport Layer 25-17

26 Custom Target Development

About Embedded Target Development **26-2**
Custom Targets **26-2**
Types of Targets **26-2**
Recommended Features for Embedded Targets **26-4**

Sample Custom Targets **26-8**

Target Development Mechanics **26-10**
Folder and File Naming Conventions **26-10**
Components of a Custom Target **26-11**
Key Folders Under Target Root (mytarget) **26-15**
Key Files in Target Folder (mytarget/mytarget) **26-18**
Additional Files for Externally Developed Targets . . . **26-20**
Target Development and the Build Process **26-21**

Customize System Target Files **26-26**
Control Code Generation With the System Target File **26-26**
System Target File Naming and Location Conventions **26-27**
System Target File Structure **26-27**
Define and Display Custom Target Options **26-35**
Tips and Techniques for Customizing Your STF **26-41**
Create a Custom Target Configuration **26-44**

Customize Template Makefiles **26-56**
Template Makefiles and Tokens **26-56**
Invoke the make Utility **26-63**
Structure of the Template Makefile **26-64**
Customize and Create Template Makefiles **26-67**

Support Optional Features **26-77**
Overview **26-77**
Support Model Referencing **26-78**
Support Compiler Optimization Level Control **26-90**
Support C Function Prototype Control **26-91**
Support C++ Class Interface Control **26-93**

Support Concurrent Execution of Multiple Tasks **26-94**

Interface to Development Tools **26-96**
About Interfacing to Development Tools **26-96**
Makefile Approach **26-97**
Interface to an Integrated Development Environment **26-97**

Device Drivers and Target Preferences **26-106**
Integrate Device Drivers **26-106**
Use Target Preferences **26-106**

Model Architecture and Design

Modeling

- “Configure a Model for Code Generation” on page 1-2
- “Scheduling” on page 1-4
- “Supported Products and Block Usage” on page 1-81
- “Modeling Semantic Considerations” on page 1-102

Configure a Model for Code Generation

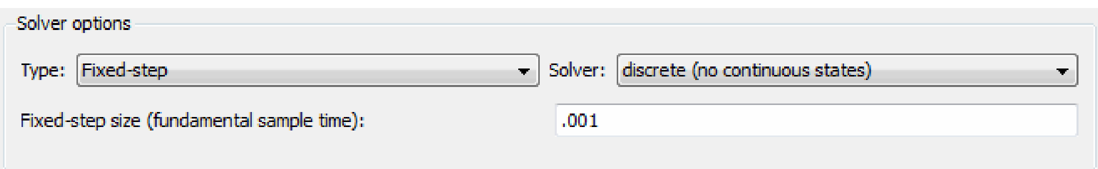
Model configuration parameters determine the method for generating the code and the resulting format.

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.

Scheduling

The following sections explain and illustrate how the Simulink® and Simulink Coder™ products handle multirate (mixed-rate) models, depending on whether code is being generated for single-tasking or multitasking environments.

In this section...
“About Scheduling” on page 1-4
“Single-Tasking and Multitasking Execution Modes” on page 1-4
“Handle Rate Transitions” on page 1-13
“Single-Tasking and Multitasking Model Execution” on page 1-26
“Handle Asynchronous Events” on page 1-31
“Timers” on page 1-69
“Configure Scheduling” on page 1-79

About Scheduling

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.

Single-Tasking and Multitasking Execution Modes

- “About Tasking Modes” on page 1-5
- “Execute Multitasking Models” on page 1-6
- “Multitasking and Pseudomultitasking Modes” on page 1-8

- "Build a Program for Multitasking Execution" on page 1-10
- "Single-Tasking Mode" on page 1-10
- "Build a Program for Single-Tasking Execution" on page 1-11
- "Model Execution and Rate Transitions" on page 1-11
- "Simulate Models with the Simulink Product" on page 1-11
- "Execute Models in Real Time" on page 1-12
- "Single-Tasking Versus Multitasking Operation" on page 1-13

About 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 **Tasking mode for periodic sample times** menu on the **Solver** pane of the Configuration Parameters dialog box. `Auto` mode (the default) applies multitasking execution for a multirate model, and otherwise selects single-tasking execution. You can also select `SingleTasking` or `MultiTasking` execution explicitly.

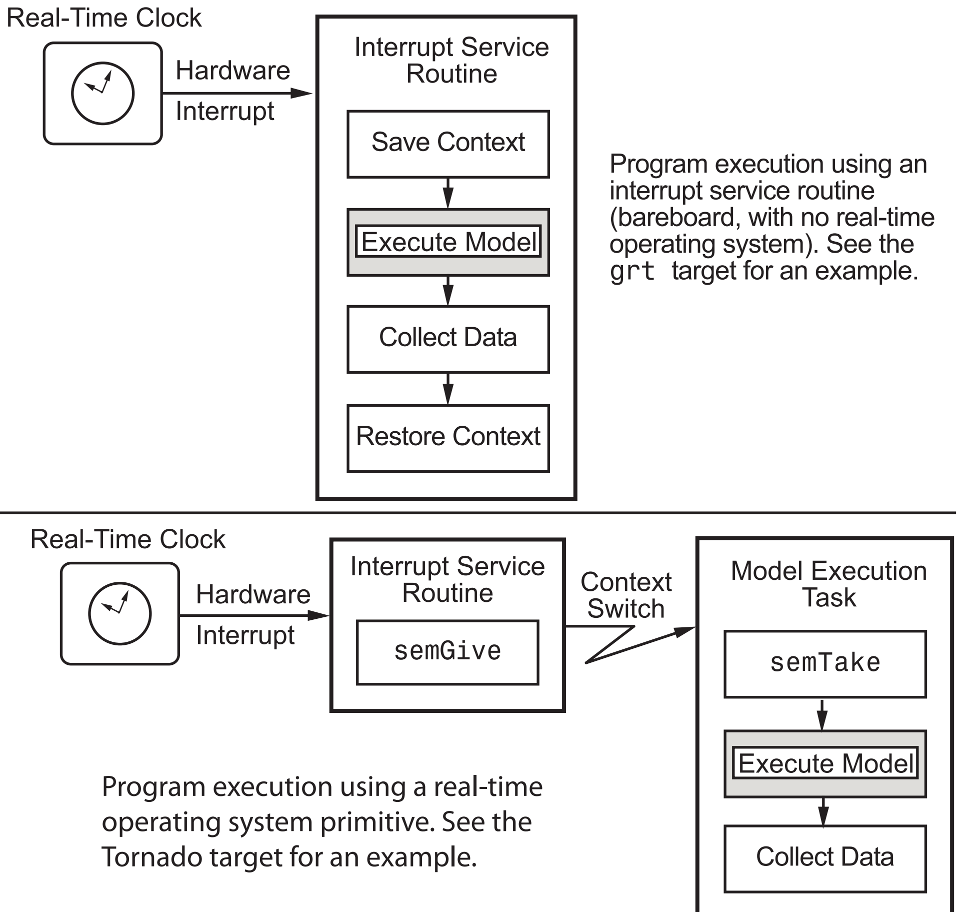
Note: A model that is multirate and uses multitasking cannot reference a multirate model that uses single-tasking.

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 a *bare-board* target, 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.



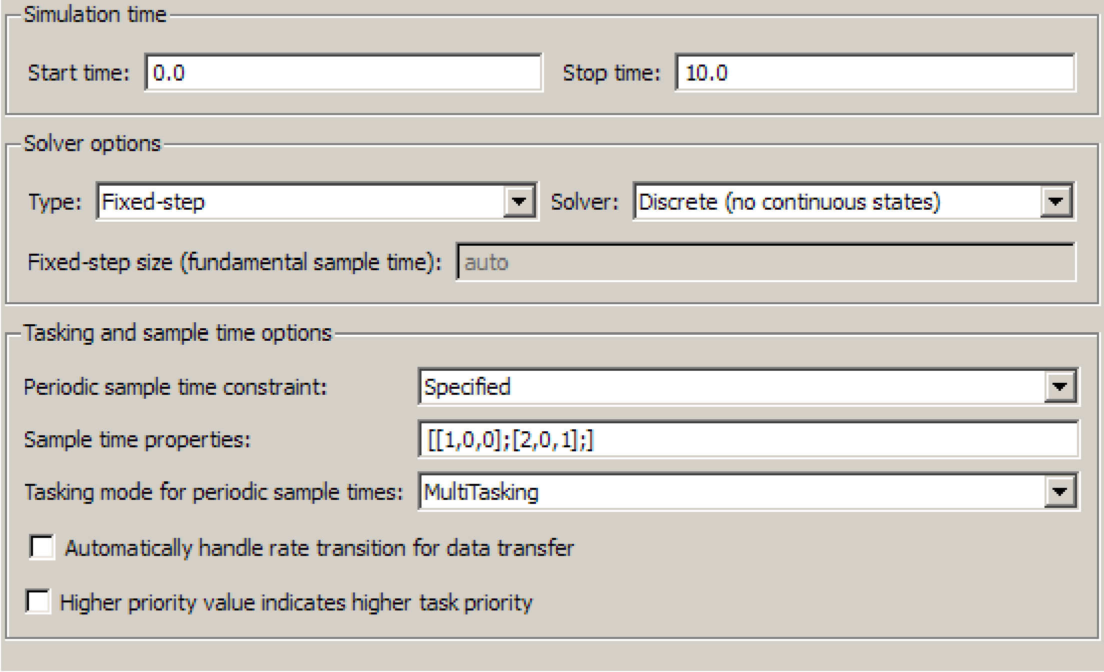
Execute Multitasking Models

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* (`tid`) 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 with the Simulink Coder software, you must

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.



- 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 and Pseudomultitasking Modes

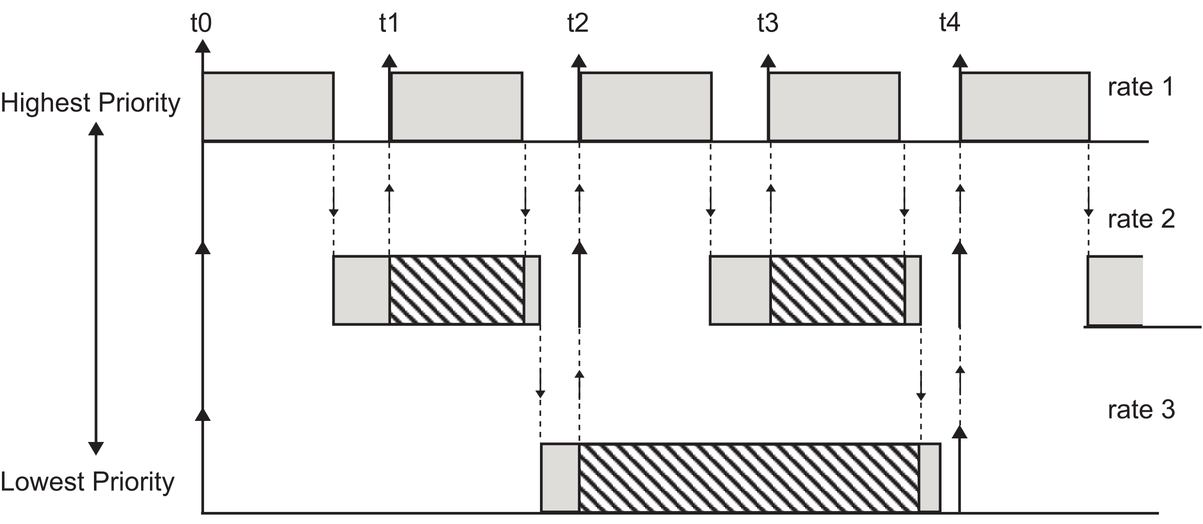
When periodic tasks execute in a multitasking mode, by default the blocks with the fastest sample rates are executed by the task with the highest priority, the next fastest blocks are executed by a task with the next higher priority, and so on. Time available in between the processing of high-priority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

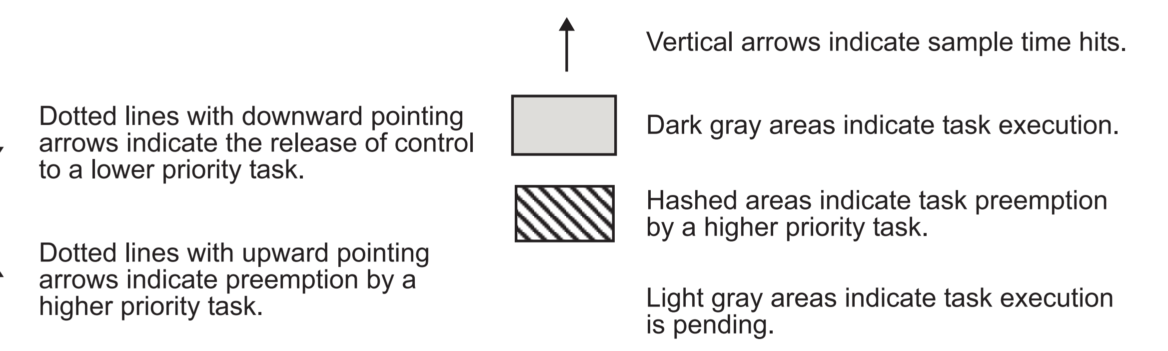
Where tasks are asynchronous rather than periodic, there may not necessarily be a relationship between sample rates and task priorities; the task with the highest priority need not have the fastest sample rate. You specify asynchronous task priorities using Async Interrupt and Task Sync blocks. You can 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. In a bare-board target (that is, no real-time operating system present), you cannot create separate tasks. However, Simulink Coder application modules implement what is effectively a multitasking execution scheme using overlapped interrupts, accompanied by programmatic context switching.

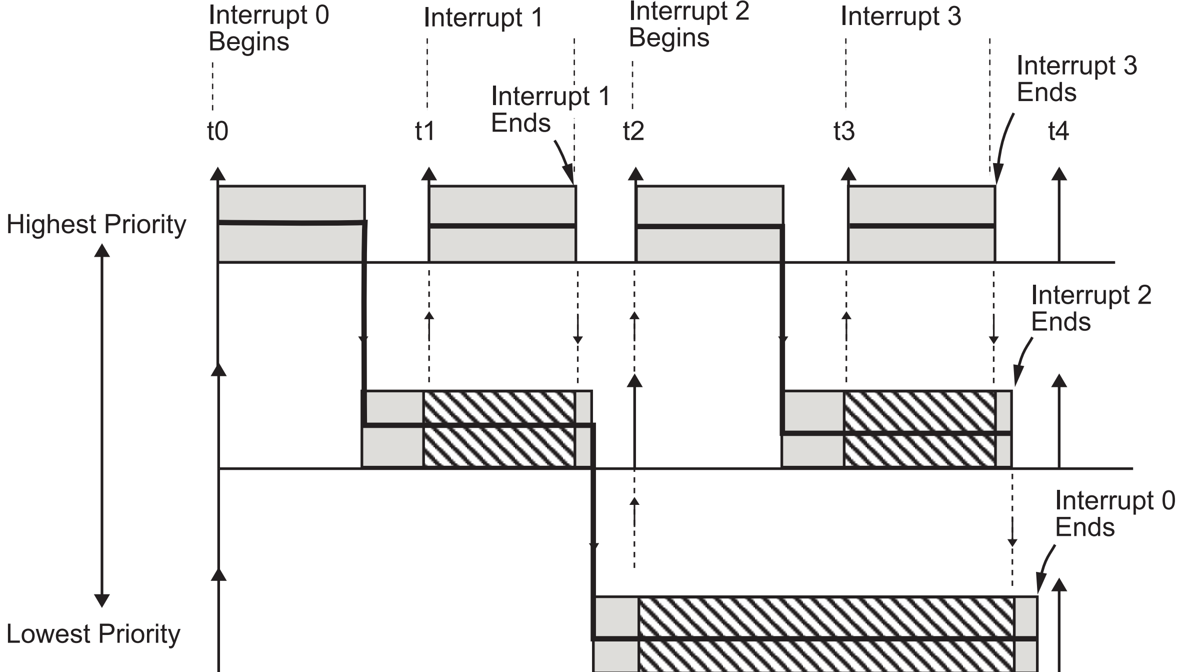
This means an interrupt can occur while another interrupt is currently in progress. When this happens, the current interrupt is preempted, the floating-point unit (FPU) context is saved, and the higher priority interrupt executes its higher priority (that is, faster sample rate) code. Once complete,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

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





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

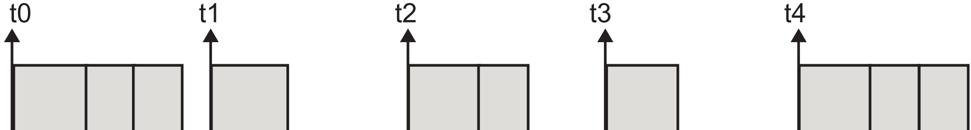
To use multitasking execution, select `Auto` (the default) or `MultiTasking` from the **Tasking mode for periodic sample times** menu on the **Solver** pane of the Configuration Parameters dialog box. This menu is active only if you select `Fixed-step` as the solver type. `Auto` mode results in a multitasking 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.

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, select `SingleTasking` from the **Tasking mode for periodic sample times** menu on the **Solver** pane of the Configuration Parameters dialog box. If you select `Auto`, single-tasking 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

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 1-19 for an explanation of this option.

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

Simulate Models with the Simulink Product

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.

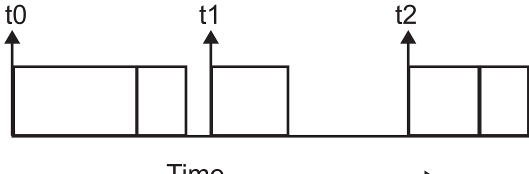
The key to this process is the ordering of blocks. A block whose output is directly dependent on its input (that is, a block with direct feedthrough) cannot execute until the block driving its input executes.

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).

Execute Models 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 `t1`. 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 1-8 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. The next section, "Handle Rate Transitions" on page 1-13, discusses issues related to rate transition blocks.

Handle Rate Transitions

- "About Rate Transitions" on page 1-13
- "Data Transfer Problems" on page 1-15
- "Data Transfer Assumptions" on page 1-16
- "Rate Transition Block Options" on page 1-16
- "Faster to Slower Transitions in a Simulink Model" on page 1-21
- "Faster to Slower Transitions in Real Time" on page 1-21
- "Slower to Faster Transitions in a Simulink Model" on page 1-23
- "Slower to Faster Transitions in Real Time" on page 1-23

About Rate Transitions

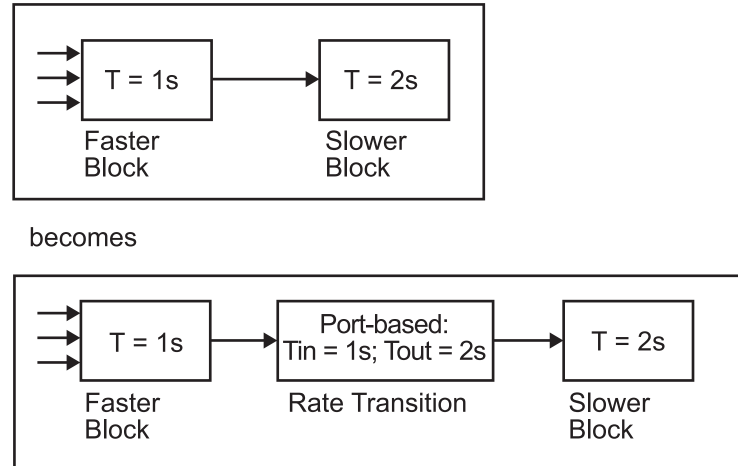
Two periodic sample rate transitions can exist within a model:

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

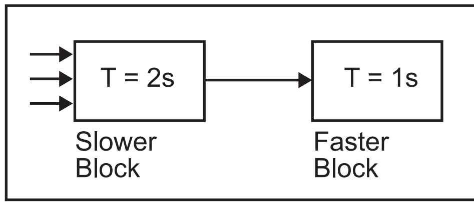
The following sections concern models with periodic sample times with zero offset only. Other considerations apply to multirate models that involve asynchronous

tasks. For details on how to generate code for asynchronous multitasking, see "Handle Asynchronous Events" on page 1-31.

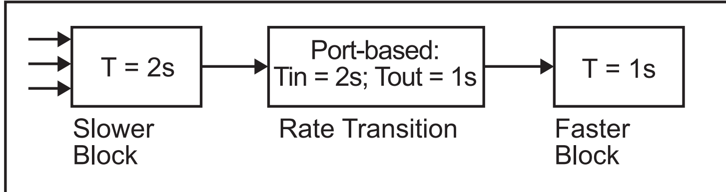
In multitasking and pseudomultitasking systems, differing sample rates can cause blocks to be executed in the wrong order. To prevent possible errors in calculated data, you must control model execution at these transitions. When connecting faster and slower blocks, you or the Simulink engine must add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



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 1-21 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 Simulink Coder software assumes the following:

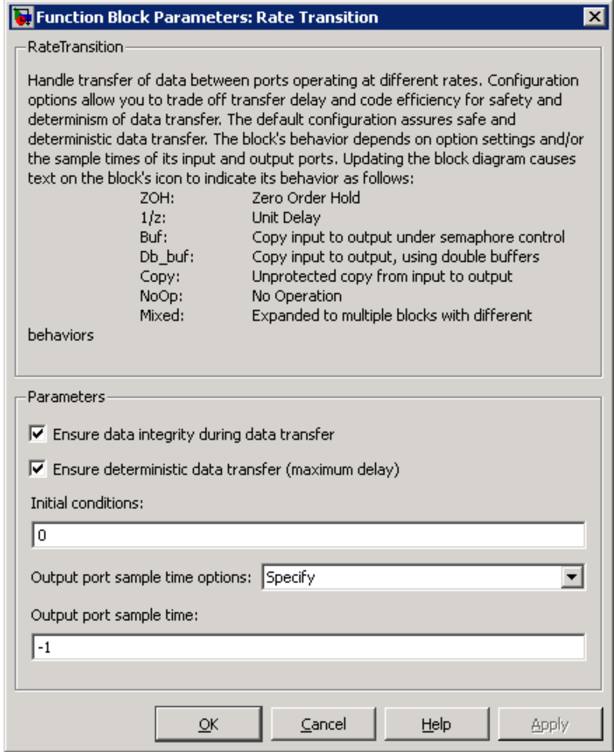
- Data transitions occur between a single reading task and a single writing task.
- A read or write of a byte-sized variable is atomic.
- When two tasks interact through a data transition, only one of them can preempt the other.
- For periodic tasks, the faster rate task has higher priority than the 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.

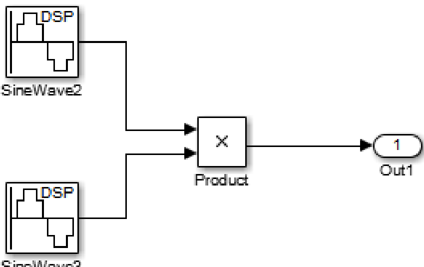
Automatic Rate Transition

The Simulink engine can detect mismatched rate transitions in a multitasking model and automatically insert Rate Transition blocks to handle them. To instruct the engine to do this, select **Automatically handle rate transition for data transfer** on the **Solver** pane of the Configuration Parameters dialog box.

The **Automatically handle rate transition for data transfer** option is off by default. When you select it,

- The Simulink engine handles transitions between periodic sample times and asynchronous tasks.
- The Simulink engine inserts "hidden" Rate Transition blocks that are not visible on the block diagram.
- The Simulink Coder software generates code for the automatically inserted Rate Transition blocks that is identical to that generated for manually inserted Rate Transition blocks.
- Automatically inserted Rate Transition blocks operate in protected mode for periodic tasks and asynchronous tasks, which you cannot alter. For periodic tasks, automatically inserted Rate Transition blocks operate with the level of determinism specified by the **Solver** pane parameter **Deterministic data transfer**. (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" in the Simulink reference documentation.) To use other modes, you must insert Rate Transition blocks and set their modes manually.

For example, in the following model **SineWave2** has a **Sample time** of 2, and **SineWave3** has a **Sample time** of 3.

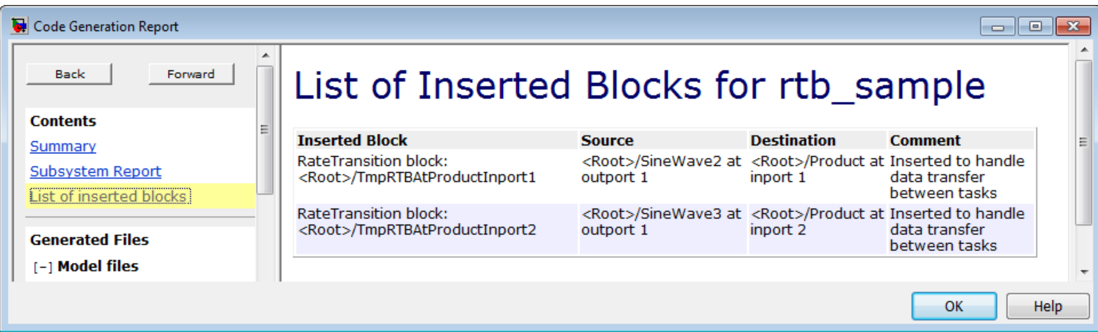


If **Automatically handle rate transition for data transfer** is on, the Simulink engine inserts an invisible Rate Transition block between each Sine Wave block and the Product block. The inserted blocks have the parameter values required to reconcile the Sine Wave block sample times.

If the input port and output port data sample rates in a model are not multiples of each other, the Simulink engine will insert 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, Simulink errors out. In this case, you must insert a Rate Transition block manually.

Inserted Rate Transition Block HTML Report

When the Simulink engine has automatically inserted Rate Transition blocks into a model, after code generation the optional HTML code generation report includes a `List of inserted blocks` that describes the blocks. For example, the following report describes the two Rate Transition blocks that the engine automatically inserts into the previous model.



Only automatically inserted Rate Transition blocks appear in a `List of inserted blocks`. If no such blocks exist in a model, the HTML code generation report does not include a `List of inserted blocks`.

Rate Transition Blocks and Continuous Time

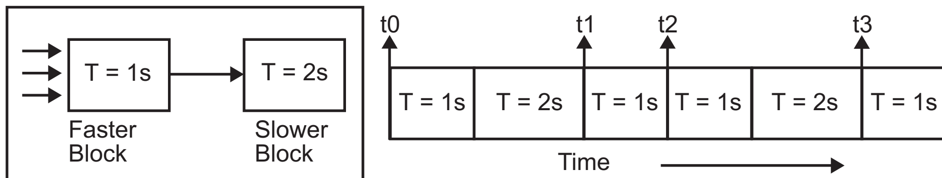
The sample time at the output port of a Rate Transition block can only be discrete or fixed in minor time step. This means that when a Rate Transition block inherits continuous sample time from its destination block, it treats the inherited sample time as Fixed in Minor Time Step. Therefore, the output 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).

The next four 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

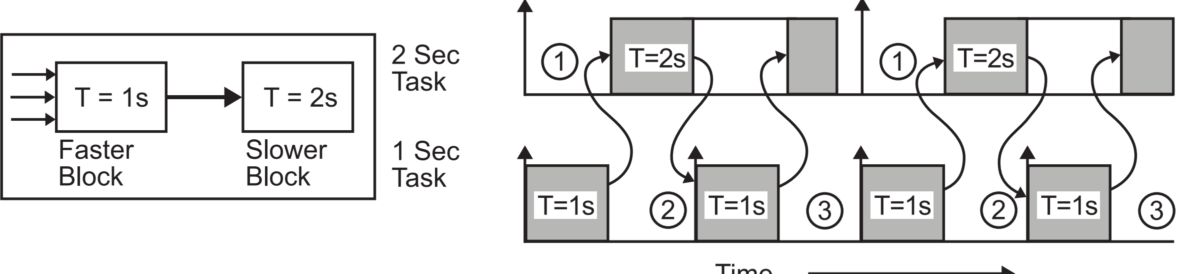
In a model where a faster block drives a slower block having direct feedthrough, the outputs of the faster block are computed first. In simulation intervals where the slower block does not execute, the simulation progresses more rapidly because there are fewer blocks to execute. The next figure illustrates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

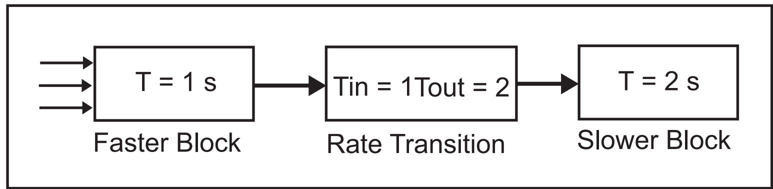
Faster to Slower Transitions in Real Time

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.



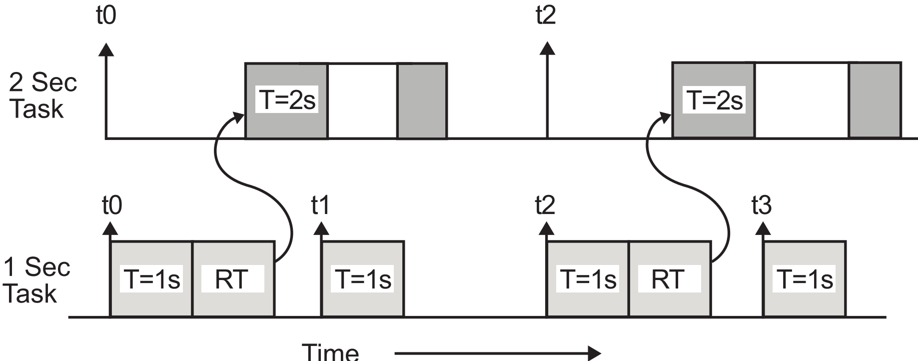
In the above figure, the faster block executes a second time before the slower block has completed execution. This can cause unpredictable results because the input data to the slow task is changing. Data might not maintain its integrity in this situation.

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.

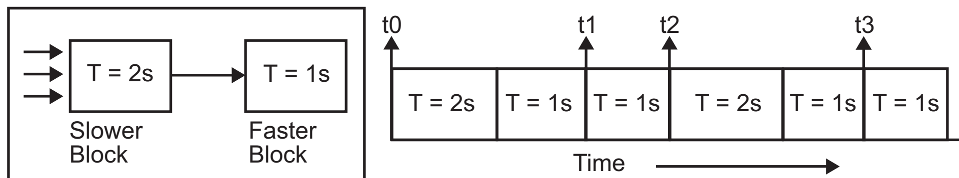


When you add a Rate Transition block, the block executes before the 2 second block (its priority is higher) and its output value is held constant while the 2 second block executes (it executes at the slower sample rate).

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.

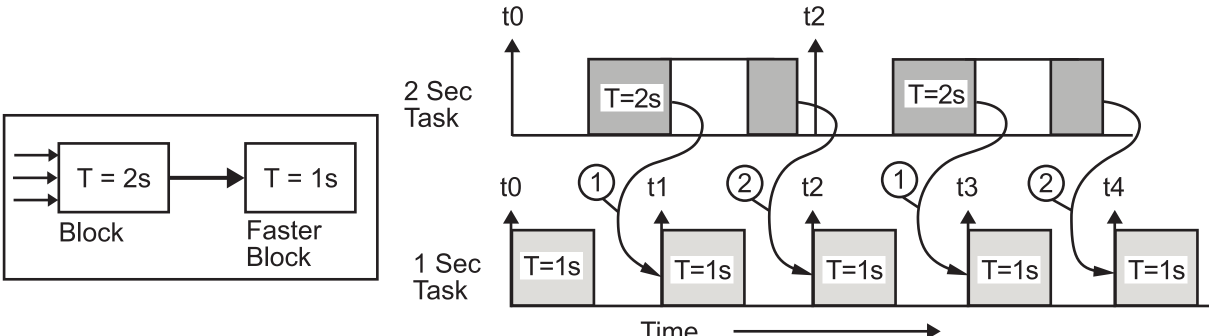
The next figure shows the execution sequence.



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

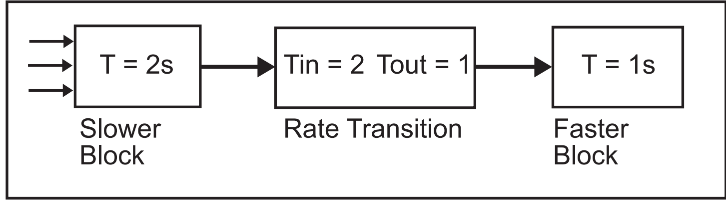
In models where a slower block drives a faster block, the generated code assigns the faster block a higher priority than the slower block. This means the faster block is executed before the slower block, which requires special care to avoid incorrect results.



This timing diagram illustrates two problems:

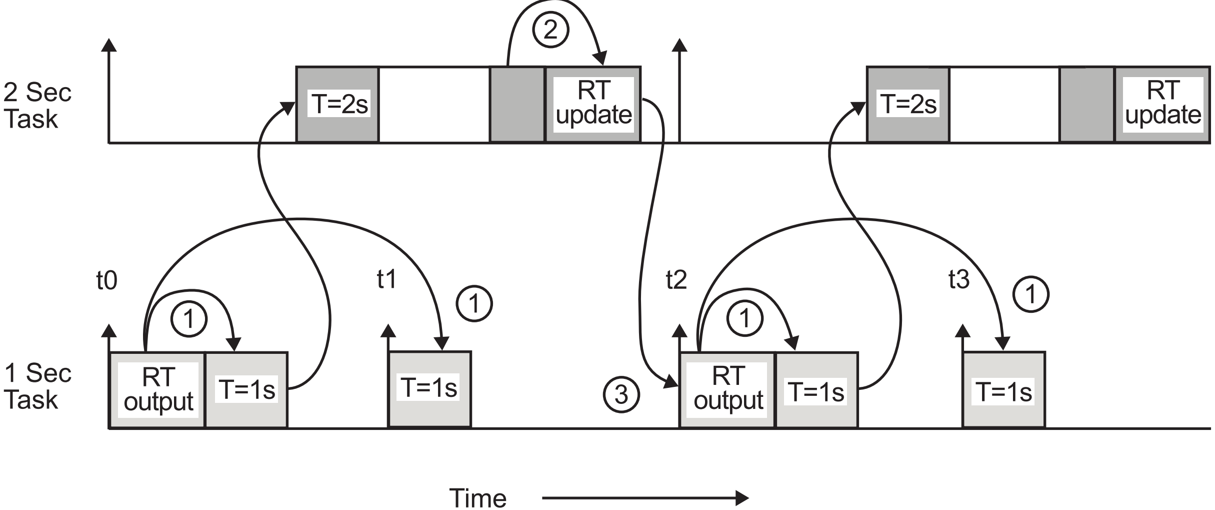
- Execution of the slower block is split over more than one faster block interval. In this case the faster task executes a second time before the slower task has completed execution. This means the inputs to the faster task can have incorrect values some of the time.
- The faster block executes before 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.

Single-Tasking and Multitasking Model Execution

- "Introduction" on page 1-26
- "Single-Tasking Execution" on page 1-27
- "Multitasking Execution" on page 1-29

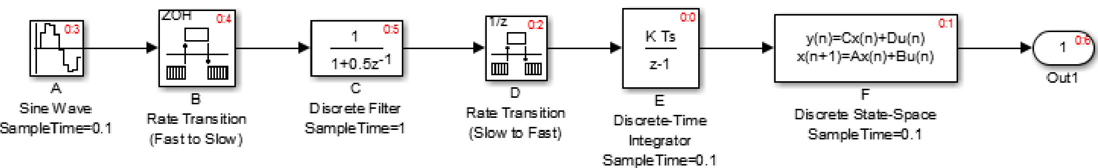
Introduction

This section examines how a simple multirate model executes in both real time and simulation, using a fixed-step solver. It considers the operation of both `SingleTasking` and `MultiTasking` **Solver** pane tasking modes.

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 "Simulating 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.

Single-Tasking Execution

This section considers the execution of the above model when the solver **Tasking mode** is `SingleTasking`.

In a single-tasking system, if the **Block reduction** option on the **Optimization** pane 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”.

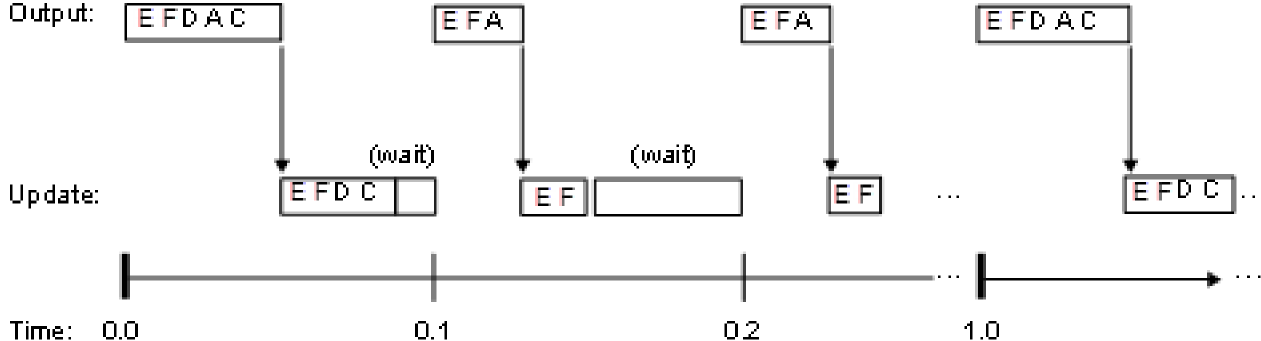
The following table shows, for each block in the model, the execution order, sample time, and whether the block has an output or update computation. Block A does not have discrete states, and accordingly does not have an update computation.

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.



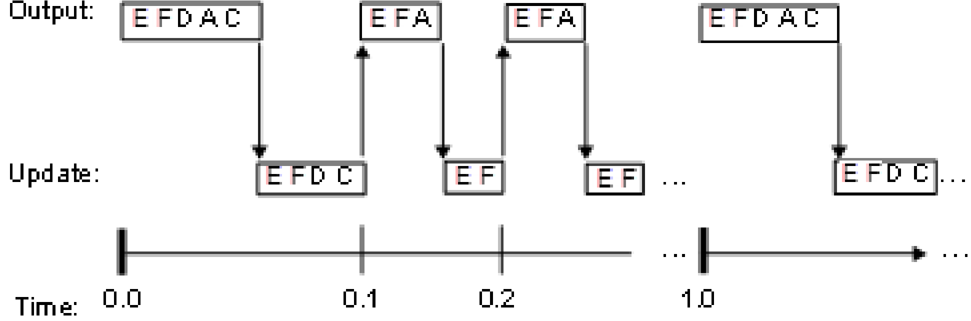
At time 0.0, 1.0, and every second thereafter, both the slow and fast blocks execute their output computations; this is followed by update computations for blocks that have states. Within a given time interval, output and update computations are sequenced in block execution order.

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.

Multitasking Execution

This section 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.

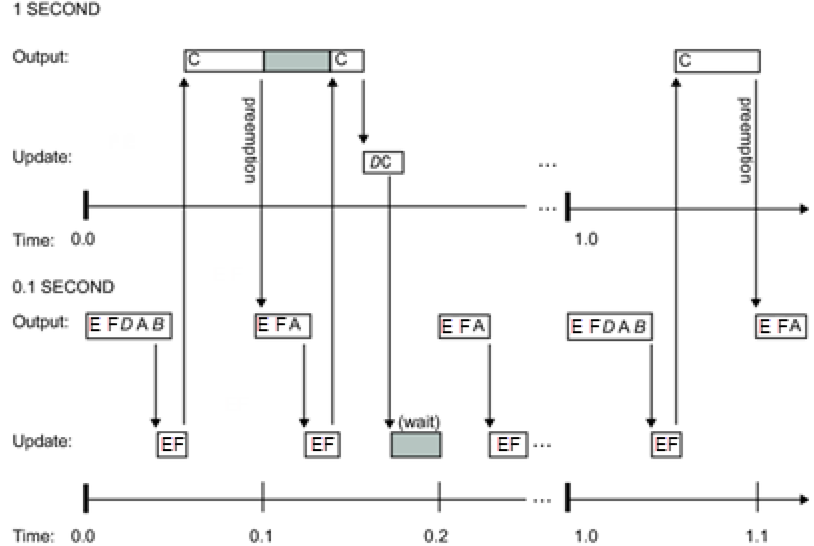
Task Allocation of Blocks in Multitasking Execution

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 “Inherit Sample Times” 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.	Y	N

Blocks (in Execution Order)	Task	Output	Update
	For more information on port-based sample times, see "Inherit Sample Times" in the Simulink documentation.		
C	1 second task	Y	Y

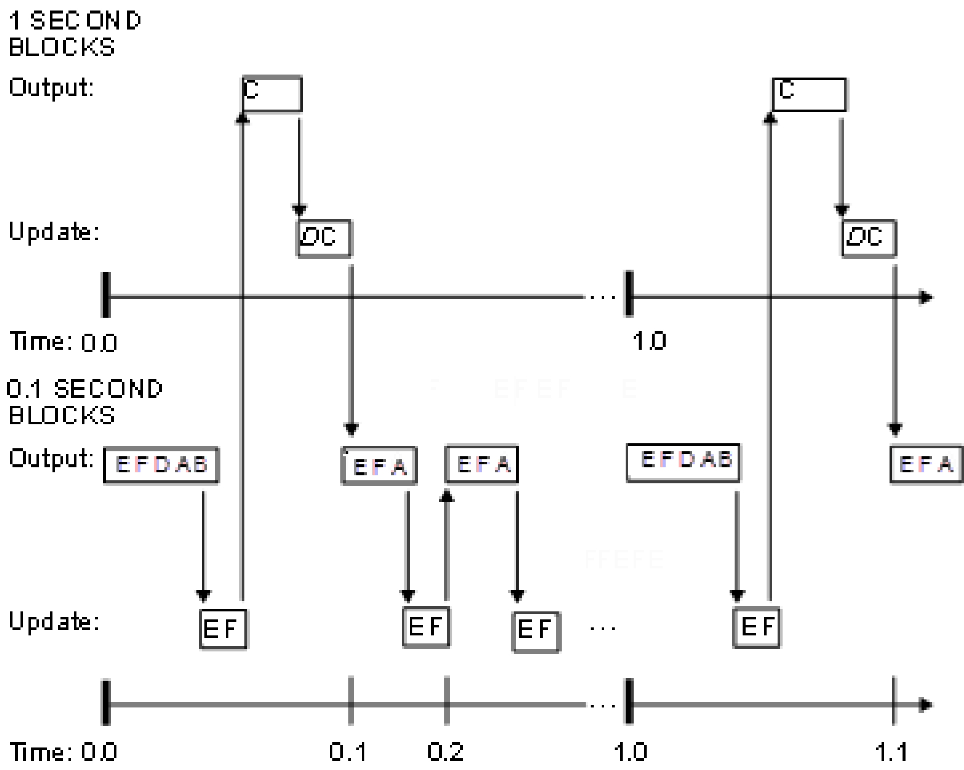
Real-Time Multitasking Execution

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.



Handle Asynchronous Events

- “About Asynchronous Events” on page 1-31
- “Handling Interrupts” on page 1-34
- “Rate Transitions and Asynchronous Blocks” on page 1-47
- “Use Timers in Asynchronous Tasks” on page 1-51
- “Create a Customized Asynchronous Library” on page 1-53
- “Import Asynchronous Event Data for Simulation” on page 1-61
- “Asynchronous Support Limitations” on page 1-64

About Asynchronous Events

- “Asynchronous Support” on page 1-32

- "Block Library for Wind River VxWorks Real-Time Operating System" on page 1-32
- "Access the VxWorks Block Library" on page 1-33
- "Generate Code with the VxWorks Library Blocks" on page 1-33
- "Examples and Additional Information" on page 1-33

Asynchronous Support

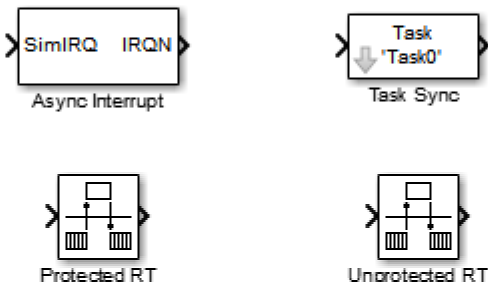
Simulink Coder models are normally timed from a *periodic* interrupt source (for example, a hardware timer). Blocks in a periodically clocked single-rate model run at a timer interrupt rate (the base rate of the model). Blocks in a periodically clocked multirate model run at the base rate or at multiples 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). Although the blocks target the Wind River® Tornado® (VxWorks® 5.x) 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.

Block Library for Wind River VxWorks Real-Time Operating System

The next figure shows the blocks in the VxWorks block library (`vxlib1`).



The key blocks in the library are the Async Interrupt and Task Sync blocks. These blocks are targeted for the Tornado (VxWorks 5.x) operating system. You can use them, without modification, to support VxWorks applications.

If you want to implement asynchronous support for an RTOS other than the VxWorks RTOS, guidelines and example code are provided that will help you to adapt the VxWorks library blocks to target your RTOS. This topic is discussed in "Create a Customized Asynchronous Library" on page 1-53.

The VxWorks library includes blocks you can use to

- Generate interrupt-level code — Async Interrupt block
- Spawn a VxWorks 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 1-47. For general information on rate transitions, see "Scheduling" on page 1-4.

Access the VxWorks Block Library

To access the VxWorks library, enter the MATLAB® command `vxlib1`.

Generate Code with the VxWorks Library Blocks

To generate a VxWorks compatible application from a model containing VxWorks library blocks, specify one of the following system target files for the model:

- `ert.tlc`. The Embedded Real-Time (ERT) target is provided with the Embedded Coder® product.

  When using the ERT target with VxWorks library blocks, you must select the **Generate an example main program** option, and select `VxWorksExample` from the **Target operating system** menu.
- `tornado.tlc`. The Tornado (VxWorks 5.x) target is provided with the Simulink Coder product.

Examples and Additional Information

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

- The `rtwdemo_async` model. To open this example, type `rtwdemo_async` at the MATLAB command prompt.
- The `rtwdemo_async_mdlreftop` model. To open this example, type `rtwdemo_async_mdlreftop` at the MATLAB command prompt.
- "Scheduling" on page 1-4, discusses general multitasking and rate transition issues for periodic models.
- The Embedded Coder documentation discusses the Embedded Real-Time (ERT) target, including task execution and scheduling.
- See your VxWorks system documentation for detailed information about the VxWorks system calls mentioned in this chapter.

Handling Interrupts

- "Generate Interrupt Service Routines" on page 1-34
- "Spawn a Wind River VxWorks Task" on page 1-41

Generate Interrupt Service Routines

To generate an interrupt service routine (ISR) associated with a specific Wind River VxWorks VME interrupt level, 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.

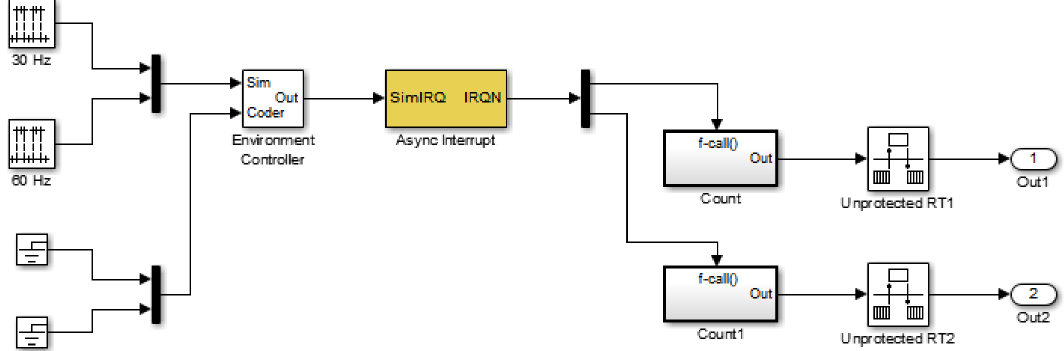
You can also use the Async Interrupt block in a simulation. It provides an input port that can be enabled and connected to a simulated interrupt source.

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 requires a VxWorks Board Support Package (BSP) that supports the following VxWorks system calls:
  - `sysIntEnable`
  - `sysIntDisable`
  - `intConnect`
  - `intLock`
  - `intUnlock`
  - `tickGet`

Performance Considerations

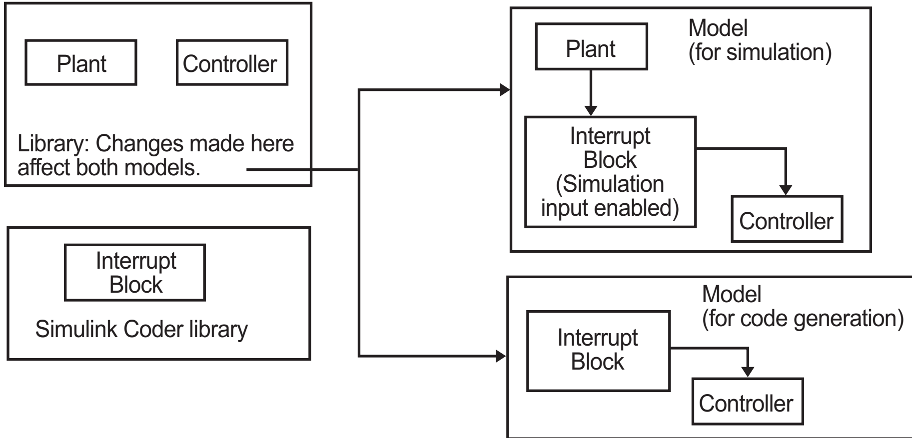
Execution of large subsystems at interrupt level can have a significant impact on interrupt response time for interrupts of equal and lower priority in the system. As a general rule, it is best to keep ISRs as short as possible. Connect only function call subsystems that contain a small number of blocks 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 VxWorks 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 VxWorks RTOS. See "Spawn a Wind River VxWorks Task" on page 1-41 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.

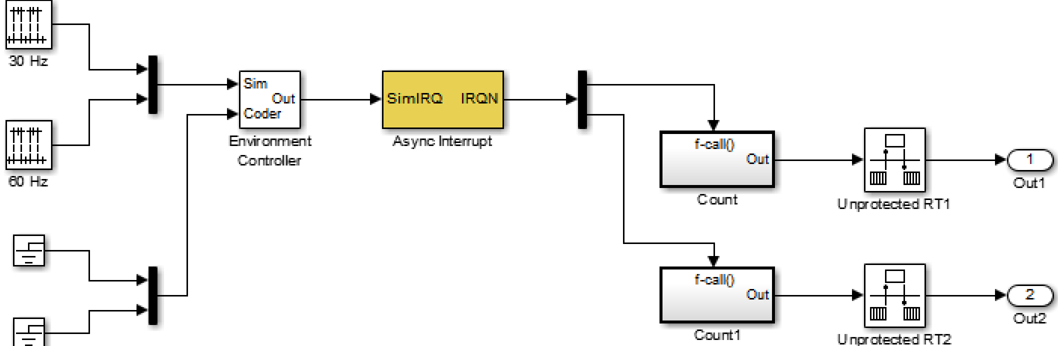


Dual-Model Use of Async Interrupt Block for Simulation and Code Generation

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 VxWorks priority. 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 VxWorks priorities. (In this case, the Async Interrupt block behaves as if all inputs received a 1 simultaneously.)

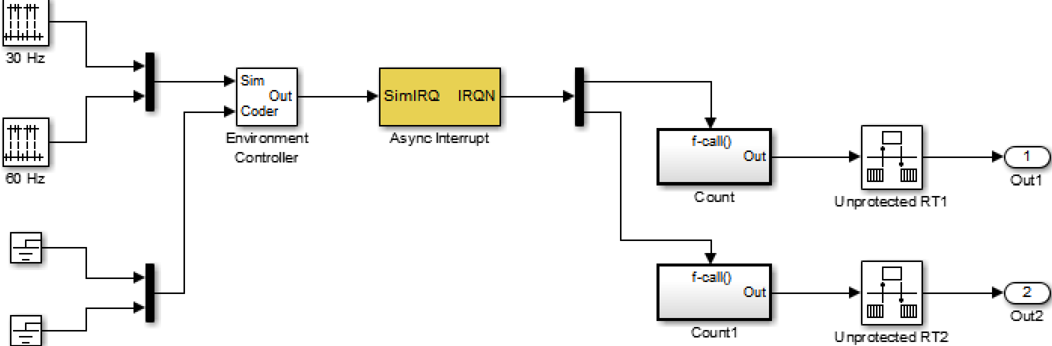
Dual-Model Approach: Code Generation

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 Simulink Coder code generator does not generate 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

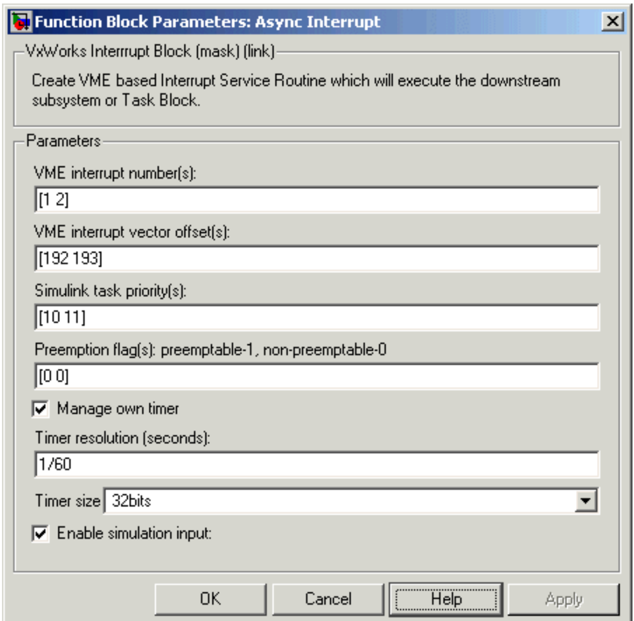
contribute to the sample times supported in the generated code. To avoid including unnecessary sample times in the generated code, use the sample times of the blocks driving the simulation input in the model where generated code is intended.



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

Offset	
192	`&isr_num1_vec192()`
193	`&isr_num2_vec193()`

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 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` 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 by the VxWorks `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 “Use Timers in Asynchronous Tasks” on page 1-51 for details.

Model Termination Code

The model's termination function disables the interrupts:

```
/* 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);
}
```

Spawn a Wind River VxWorks Task

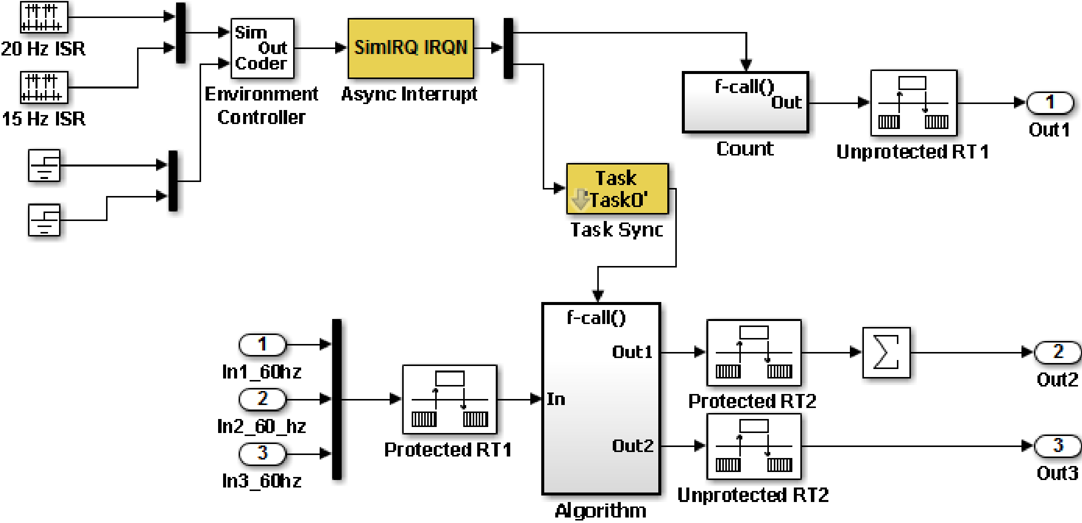
To spawn an independent VxWorks task, use the Task Sync block. The Task Sync block is a function call subsystem that spawns an independent VxWorks task. The task calls the function call subsystem connected to the output of the Task Sync block.

Typically, the Task Sync block is placed between an Async Interrupt block and a function call subsystem block or a Stateflow chart. Another example would be to place the Task Sync block at the output of a Stateflow chart that has an event, `Output to Simulink`, configured as a function call.

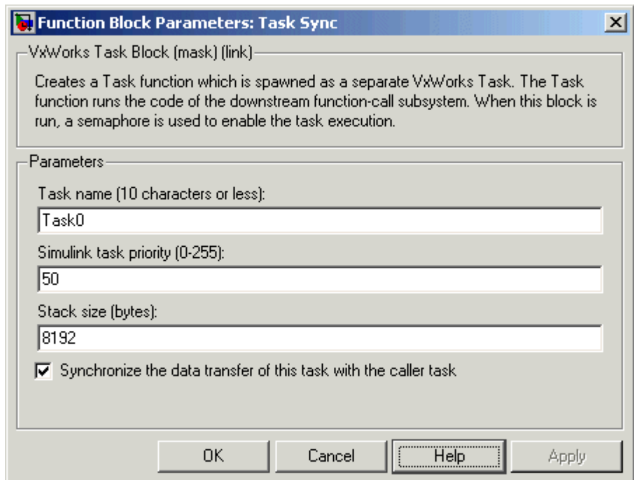
The Task Sync block performs the following functions:

- An independent task is spawned, using the VxWorks system call `taskSpawn`. When the task is activated, it calls the downstream function call subsystem code. The task is deleted using `taskDelete` during model termination.
- A semaphore is created to synchronize the connected subsystem to the execution of the Task Sync block.
- The spawned task is wrapped in an infinite `for` loop. In the loop, the spawned task listens for the semaphore, using `semTake`. When `semTake` is first called, `NO_WAIT` is specified. This allows the task to determine whether a second `semGive` has occurred prior to the completion of the function call subsystem. This would indicate that the interrupt rate is too fast or the task priority is too low.
- The Task Sync block generates synchronization code (for example, `semGive()`). This code allows the spawned task to run; the task in turn calls the connected function call subsystem code. The synchronization code can run at interrupt level. This is accomplished by connecting the Task Sync block to the output of an Async Interrupt block, which triggers execution of the Task Sync block within an ISR.
- If blocks in the downstream algorithmic code require absolute time, it can be supplied either by the timer maintained by the Async Interrupt block, or by an independent timer maintained by the task associated with the Task Sync block.

For an example of how to use the Task Sync block, see the `rtwdemo_async` example. The block diagram for the model appears in the next figure. Before reading the following discussion, open the example model and double-click the **Generate Code** button. You can then examine the generated code in the HTML code generation report produced by the example.



In this model, the Async Interrupt block is configured for VME interrupts 1 and 2, using interrupt vector offsets 192 and 193. Interrupt 2 is connected to the Task Sync block, which in turn drives the Algorithm subsystem. Consider the code generated from this model, assuming that the Task Sync block parameters are configured as shown in the next figure.



Initialization Code

The Task Sync block generates initialization code for initialization by `MdlStart`, which itself creates and initializes the synchronization semaphore. It also spawns an independent task (`task0`).

```
/* VxWorks Task Block: <S5>/S-Function (vxtask1) */
/* Spawn task: Task0 with priority 50 */
if ((*(SEM_ID *)rtwdemo_async_DWork.SFunction_PWORK.SemID =
  semBCreate(SEM_Q_PRIORITY, SEM_EMPTY)) == NULL) {
  printf("semBCreate call failed for block Task0.\n");
}
if ((rtwdemo_async_DWork.SFunction_IWORK.TaskID = taskSpawn("Task0",
   50.0, VX_FP_TASK, 8192.0, (FUNCPTR)Task0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0)) == ERROR) {
  printf("taskSpawn call failed for block Task0.\n");
}
```

After spawning `Task0`, `MdlStart` connects and enables the 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);
```

The ordering of these operations is significant. The task must be spawned before the interrupt that activates it can be enabled.

Task and Task Synchronization Code

The function `Task0`, generated by the Task Sync block, runs as a VxWorks task. The task waits for a synchronization semaphore in an infinite `for` loop. If it obtains the semaphore, it updates its task timer and calls the Algorithm subsystem.

For this example, the **Synchronize the data transfer of this task with the caller task** option of the Task Sync block is selected. Therefore, the timer associated with the Task Sync block (`rtM->Timing.clockTick2`) is updated with the value of the timer that is maintained by the Async Interrupt block (`rtM->Timing.clockTick3`). Therefore, blocks within the Algorithm subsystem use timer values based on the time of the most recent interrupt (not the most recent activation of `Task0`).

```
/* VxWorks Task Block: <S5>/S-Function (vxtask1) */
/* Spawned with priority: 50 */
void Task0(void)
{
  /* Wait for semaphore to be released by system:
     rtwdemo_async/Task Sync */
  for(;;) {
    if (semTake(*(SEM_ID
      *)rtwdemo_async_DWork.SFunction_PWORK.SemID,NO_WAIT) !=
```

```
      ERROR) {
        logMsg("Rate for Task Task0() too fast.\n",0,0,0,0,0,0);
#if STOPONOVERRUN
      logMsg("Aborting real-time simulation.\n",0,0,0,0,0,0);
      semGive(stopSem);
      return(ERROR);
#endif
    } else {
      semTake(*(SEM_ID
      *)rtwdemo_async_DWork.SFunction_PWORK.SemID,
      WAIT_FOREVER);
    }
    /* Use the upstream clock tick counter for this Task. */
    rtwdemo_async_M->Timing.clockTick2 =
    rtwdemo_async_M->Timing.clockTick3;

    /* Call the system: <Root>/Algorithm */
{

      /* Output and update for function-call system: '<Root>/Algorithm' */

      {

        uint32_T rt_currentTime = ((uint32_T)rtwdemo_async_M->Timing.clockTick2);
        uint32_T rt_elapseTime = rt_currentTime -
          rtwdemo_async_DWork.Algorithm_PREV_T;
        rtwdemo_async_DWork.Algorithm_PREV_T = rt_currentTime;

        {
          int32_T i;

          /* DiscreteIntegrator: '<S1>/Integrator' */
          rtwdemo_async_B.Integrator = rtwdemo_async_DWork.Integrator_DSTATE;
          for(i = 0; i < 60; i++) {

            /* Sum: '<S1>/Sum' */
            rtwdemo_async_B.Sum[i] = rtwdemo_async_B.ProtectedRT1[i] + 1.25;
          }
        }

        /* Sum: '<S1>/Sum1' */
        rtwdemo_async_B.Sum1 = rtwdemo_async_B.Sum[0];
        {
          int_T i1;

          const real_T *u0 = &rtwdemo_async_B.Sum[1];

          for (i1=0; i1 < 59; i1++) {
            rtwdemo_async_B.Sum1 += u0[i1];
          }
        }

        {
          int32_T i;
          if(rtwdemo_async_DWork.ProtectedRT2_ActiveBufIdx) {
            for(i = 0; i < 60; i++) {
              rtwdemo_async_DWork.ProtectedRT2_Buffer0[i] =
                rtwdemo_async_B.Sum[i];
            }
            rtwdemo_async_DWork.ProtectedRT2_ActiveBufIdx = (boolean_T)0U;
          } else {
            for(i = 0; i < 60; i++) {
```

```
            rtwdemo_async_DWork.ProtectedRT2_Buffer1[i] =
              rtwdemo_async_B.Sum[i];
          }
          rtwdemo_async_DWork.ProtectedRT2_ActiveBufIdx = (boolean_T)1U;
        }
      }

      /* Update for DiscreteIntegrator: '<S1>/Integrator' */
      rtwdemo_async_DWork.Integrator_DSTATE = (real_T)rt_elapseTime *
        1.6666666666666666E-002 * rtwdemo_async_B.Sum1 +
        rtwdemo_async_DWork.Integrator_DSTATE;
    }
```

The semaphore is granted by the function `isr_num2_vec193`, which is called from interrupt level:

```
/* 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 */
  rtwdemo_async_M->Timing.clockTick3 = tickGet();

  /* Call the system: <S4>/Subsystem */

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

    {
      int32_T i;
      for(i = 0; i < 60; i++) {
        if(rtwdemo_async_DWork.ProtectedRT1_ActiveBufIdx) {
          rtwdemo_async_B.ProtectedRT1[i] =
            rtwdemo_async_DWork.ProtectedRT1_Buffer1[i];
        } else {
          rtwdemo_async_B.ProtectedRT1[i] =
            rtwdemo_async_DWork.ProtectedRT1_Buffer0[i];
        }
      }
    }

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

The ISR maintains a timer that stores the tick count at the time of interrupt. This timer is updated before releasing the semaphore that activates `Task0`.

As this example shows, the Task Sync block generates only a small amount of interrupt-level code.

Task Termination Code

The Task Sync block also generates the following termination code.

```
/* Model terminate function */

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 function-call system: '<S4>/Subsystem' */
  /* VxWorks Task Block: <S5>/S-Function (vxtask1) */
  /* Destroy task: Task0 */
  taskDelete(rtwdemo_async_DWork.SFunction_IWORK.TaskID);
  }
```

Rate Transitions and Asynchronous Blocks

- “About Rate Transitions and Asynchronous Blocks” on page 1-47
- “Handle Rate Transitions for Asynchronous Tasks” on page 1-49
- “Handle Multiple Asynchronous Interrupts” on page 1-49

About 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 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 “Scheduling” on page 1-4.

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 Simulink Coder software 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 Simulink Coder software 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 Wind River VxWorks RTOS 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 a VxWorks priority to the generated task code).

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

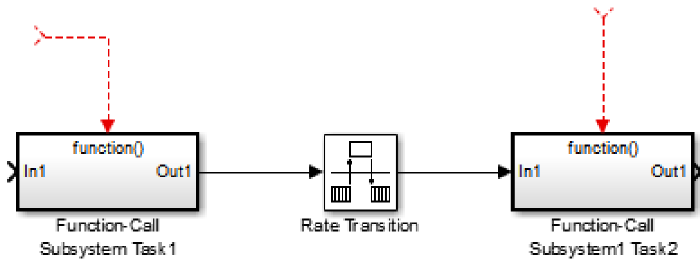
- Protected Rate Transition block: Rate Transition block that is configured with the **Ensure data integrity during data transfers** on and **Ensure deterministic data transfer** off.

- 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:

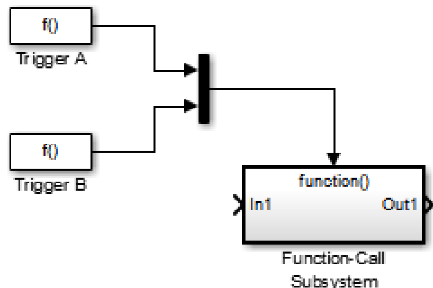
- Maintain data integrity, no determinism
- Unprotected

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

- 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 Simulink Coder product 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 Simulink Coder code generator generates 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 "Use Timers in Asynchronous Tasks" on page 1-51 and "Control Memory Allocation for Time Counters" on page 18-8 for more information.

Use Timers in Asynchronous Tasks

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

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 library blocks for the Wind River VxWorks RTOS 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 VxWorks kernel. The maximum word size for the timer is UINT32. The following VxWorks example for the 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 targeting the VxWorks RTOS, you can obtain better timer resolution by replacing the `tickGet` call and accessing a hardware timer by using your BSP instead.

If you are implementing a custom asynchronous block for an RTOS other than the VxWorks RTOS, 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 1-54.

Create a Customized Asynchronous Library

- "About Implementing Asynchronous Blocks" on page 1-53
- "Async Interrupt Block Implementation" on page 1-54
- "Task Sync Block Implementation" on page 1-58
- "asynclib.tlc Support Library" on page 1-60

About Implementing Asynchronous Blocks

This section 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.)

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 Wind River VxWorks RTOS 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 “Insert S-Function Code” 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`:

- `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 1-60.

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_ASYNCHRONOUS_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_ASYNCHRONOUS` in *`matlabroot`*`/simulink/include/simstruc.h`.

```
ssSetOptions( S, (SS_OPTION_EXCEPTION_FREE_CODE |
              SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME |
              SS_OPTION_ASYNCHRONOUS_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 VxWorks header files. 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.
- 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 a 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.

If you are targeting the VxWorks RTOS, you can obtain better timer resolution by replacing the `tickGet` call and accessing a hardware timer by using your BSP instead. `tickGet` supports only a `1/60` second resolution.

Start Function

The `Start` function generates the required 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`. They are

- `vxtask1.c`: 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 1-60.

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 VxWorks header files. 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 identifier strings 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 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 code format 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 a VxWorks task (taskSpawn). You should replace these with calls to your target RTOS.

Outputs Function

The Outputs function generates a 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 a Wind River VxWorks Task” on page 1-41. Outputs also generates code (called from interrupt level) that grants the semaphore.

Terminate Function

The Terminate function generates the 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
LibBlockExecuteFcnCall	For use by inlined S-functions with function call outputs. Generates code to execute a function call subsystem.
LibGetBlockAttribute	Returns a field value from a block record.
LibGetFcnCallBlock	Given an S-Function block and call index, returns the block record for the downstream function call subsystem block.
LibGetCallerClockTickCounter	Provides access to the time counter of an upstream asynchronous task.
LibGetCallerClockTickCounter-HighWord	Provides access to the high word of the time counter of an upstream asynchronous task.
LibManageAsyncCounter	Determines whether an asynchronous task needs a counter and manages its own timer.
LibNeedAsyncCounter	If the calling block requires an asynchronous counter, returns TLC_TRUE, otherwise returns TLC_FALSE.

Function	Description
`LibSetAsyncClockTicks`	Returns code that sets `clockTick` counters that are to be maintained by the asynchronous task.
`LibSetAsyncCounter`	Generates code to set the tick value of the block's asynchronous counter.
`LibSetAsyncCounterHighWord`	Generates code to set the tick value of the high word of the block's asynchronous counter

Import Asynchronous Event Data for Simulation

Capabilities

You can inport 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

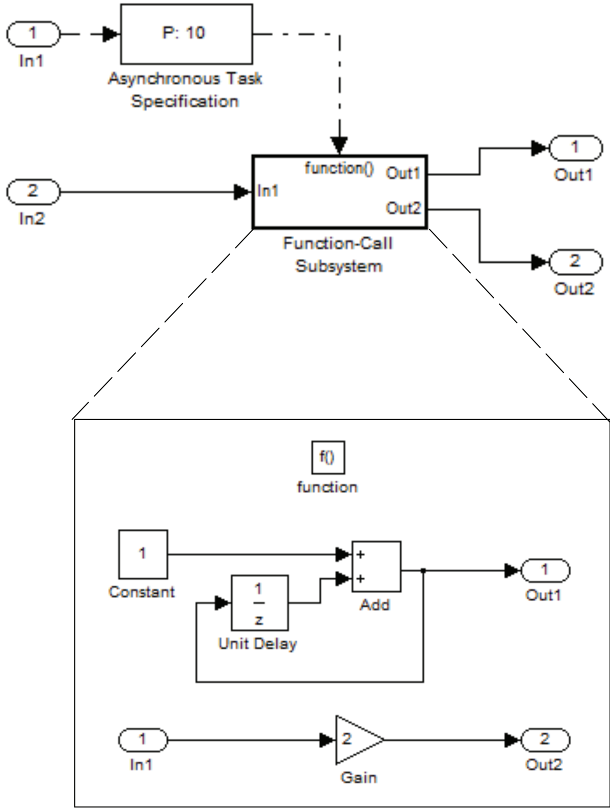
Input Data Format

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, as described in “Enable Data Import”.
- 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 as described in “Enable Data Import”.

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,

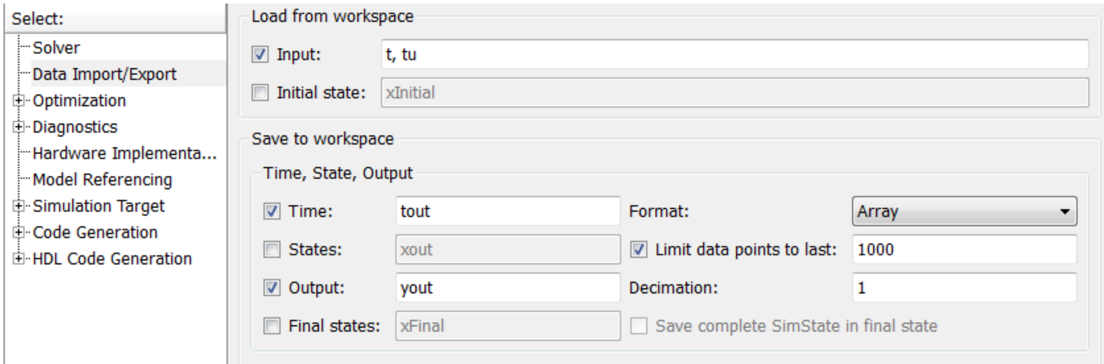
   a Select **Simulation > Configuration Parameters > Data Import/Export**.

   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

Asynchronous Support Limitations

- "Asynchronous Task Priority" on page 1-64
- "Convert an Asynchronous Subsystem into a Model Reference" on page 1-64

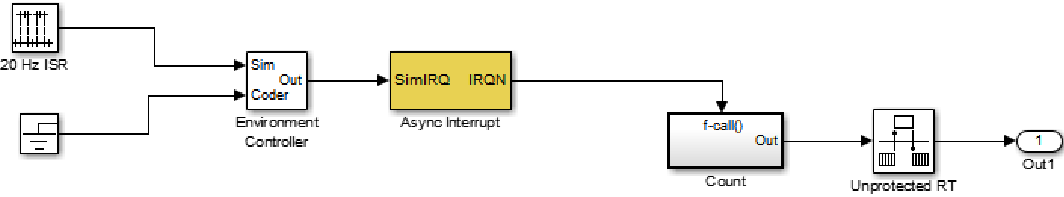
Asynchronous Task Priority

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.

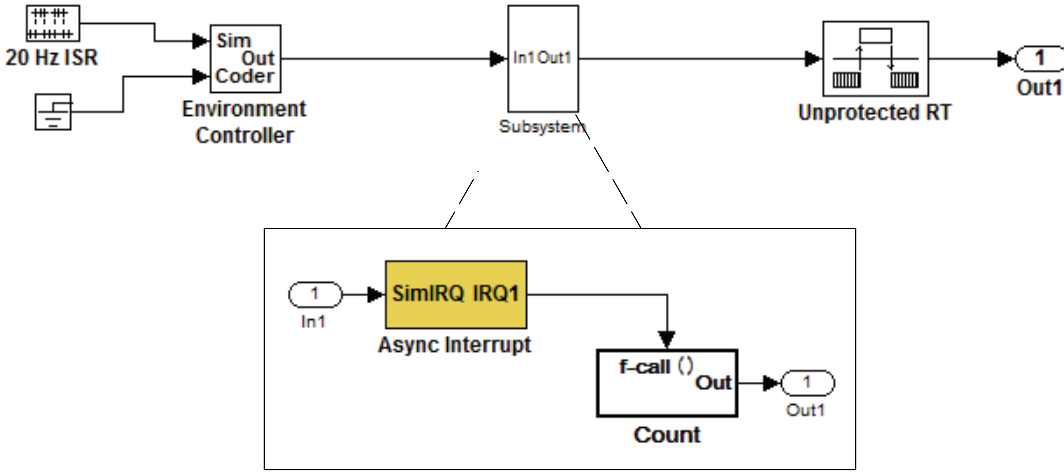
Convert an Asynchronous Subsystem into a Model Reference

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.



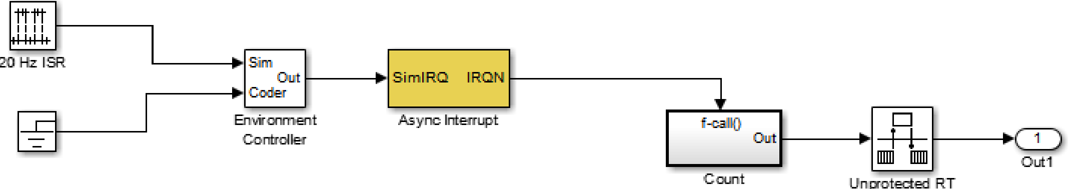
1. Convert the Async Interrupt and Count blocks into a subsystem. Select both blocks and right-click Count. From the menu, select **Subsystem & Model Reference** > **Create Subsystem from Selection**.

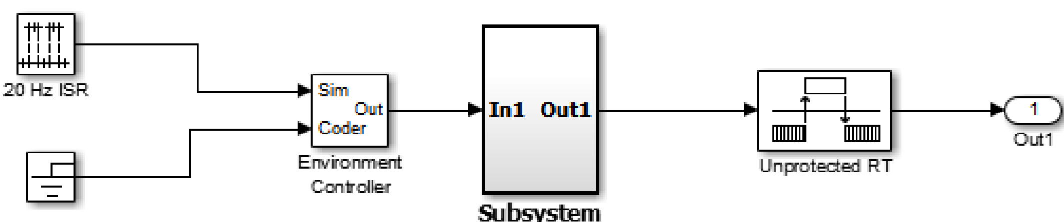


2 To prepare for converting the new subsystem to a Model block, set the following configuration parameters in the top model. Open the Configuration Parameters dialog box.

- If you are simulating in Normal mode, then you must make the following change. From the Optimization node, navigate to the Signals and Parameters pane. Under **Simulation and code generation**, select the **Inline parameters** option.
- From the Diagnostics node, 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 Data Validity pane and set the **Multitask data store** option to `error` and set the **Underspecified initialization detection** to `Simplified`. If your model is large or complex, in the Model Advisor, run the **Check consistency of initialization parameters for Outport and Merge blocks** check and make suggested changes.
- 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`.

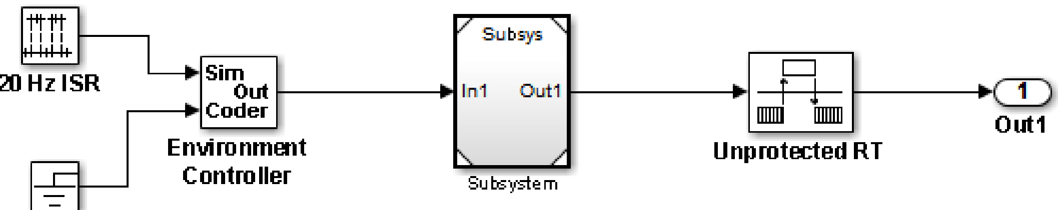
3 Convert the subsystem to an atomic subsystem. Select **Edit > Subsystem Parameters > Treat as atomic unit**.



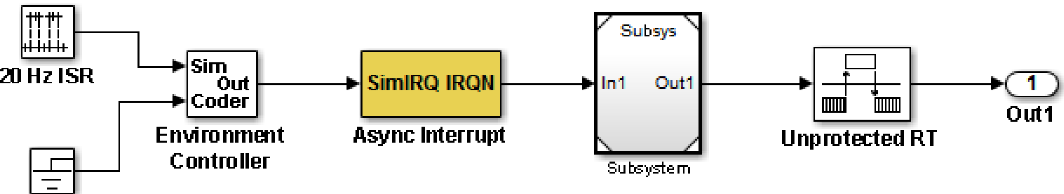


4 Convert the subsystem to a Model block. Right-click the subsystem and select **Subsystem & Model Reference** > **Convert Subsystem to** > **Referenced Model**. A window opens with a model reference block inside of it.

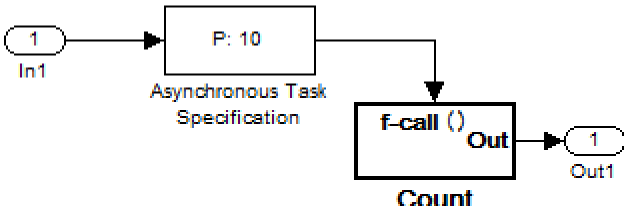
5 Replace the subsystem in the top model with the new model reference block.



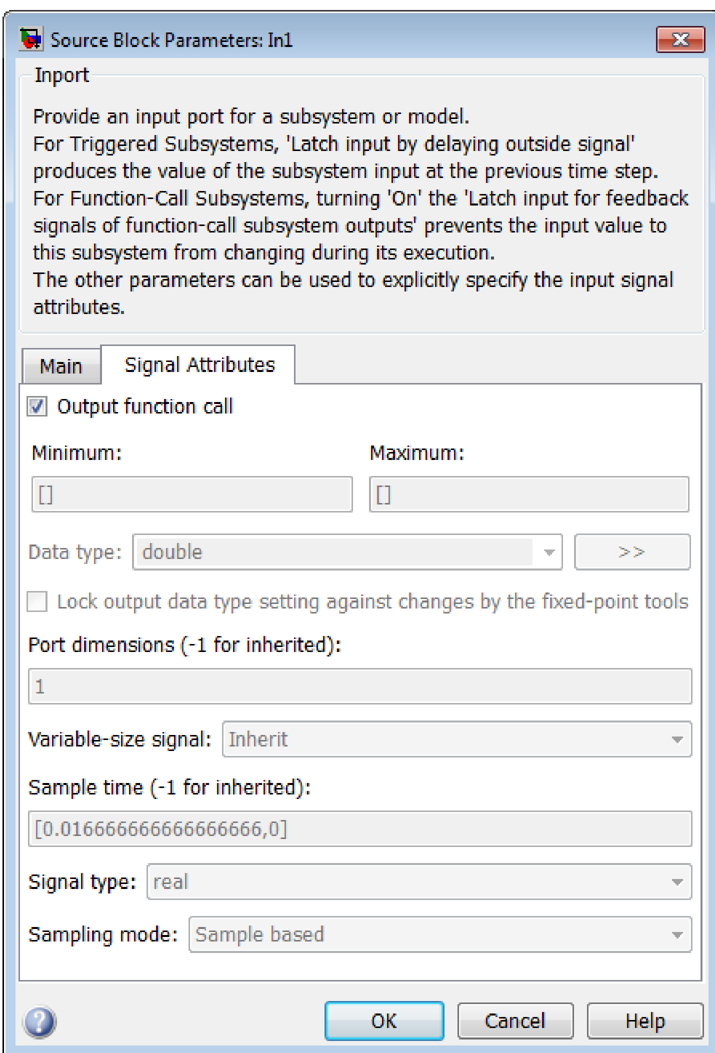
6 Move the Async Interrupt block from the model reference to the top model, before the model reference block.



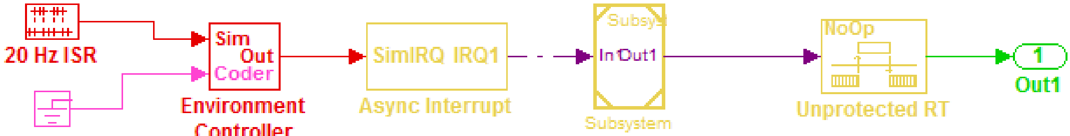
7 Insert an Asynchronous Task Specification block in the model reference. Set the priority of the Asynchronous Task Specification block. (For more information on setting the priority, see "Asynchronous Task Specification".)



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.



Timers

- “Absolute and Elapsed Time Computation” on page 1-69
- “APIs for Accessing Timers” on page 1-71
- “Elapsed Timer Code Generation Example” on page 1-75
- “Limitations on the Use of Absolute Time” on page 1-78

Absolute and Elapsed Time Computation

- “About Timers” on page 1-69
- “Timers for Periodic and Asynchronous Tasks” on page 1-70
- “Allocation of Timers” on page 1-70
- “Integer Timers in Generated Code” on page 1-70
- “Elapsed Time Counters in Triggered Subsystems” on page 1-71

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.

- The Simulink Coder product provides S-function and TLC APIs that let your S-functions access timers, in both 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".

See "Limitations on the Use of Absolute Time" on page 1-78 and "Blocks that Depend on Absolute Time" on page 1-78 for more information about absolute time and the restrictions that it imposes.

Timers for Periodic and Asynchronous Tasks

This chapter discusses timing services provided for blocks executing within *periodic* tasks (that is, tasks running at the model's base rate or subrates).

The Simulink Coder product also provides timer support for blocks whose execution is *asynchronous* with respect to the periodic timing source of the model. See the following sections of the Asynchronous Support chapter:

- "Use Timers in Asynchronous Tasks" on page 1-51
- "Create a Customized Asynchronous Library" on page 1-53

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 "APIs for Accessing Timers" on page 1-71). 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 “Use Timers in Asynchronous Tasks” on page 1-51 and “Control Memory Allocation for Time Counters” on page 18-8 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 “APIs for Accessing Timers” on page 1-71). 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.

APIs for Accessing Timers

- “About Timer APIs” on page 1-71
- “C API for S-Functions” on page 1-72
- “TLC API for Code Generation” on page 1-74

About Timer APIs

This section describes APIs that let your S-functions take advantage of the efficiencies offered by the 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 will continue to operate as expected, but will generate old-style, less efficient code.

C API for S-Functions

The SimStruct macros described in this section 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 1-71.
- `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 elaspe 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.

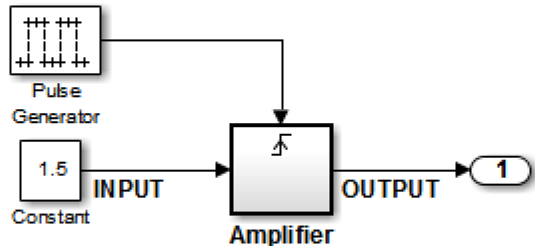
  Note Do not use `LibGetT` for this purpose. `LibGetT` always reads the base rate (`tid 0`) timer. If `LibGetT` is called for a block executing at a subrate, the wrong timer is read, causing serious errors.

- `LibGetElapseTime(system)`: Generates code to read the elapsed time counter for `system`. (`system` is the parent system of the calling block.) See “Elapsed Timer Code Generation Example” on page 1-75 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.)

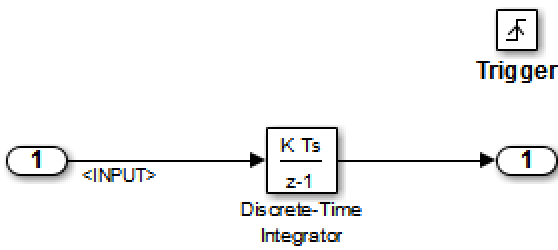
- `LibGetElapseTimeCounterDtypeId(system)`: Generates code that returns the data type of the elapsed time counter for `system`. (`system` is the parent system of the calling block.)
- `LibGetElapseTimeResolution(system)`: Generates code that returns the resolution of the elapsed time counter for `system`. (`system` is the parent system of the calling block.)

Elapsed Timer Code Generation Example

This section shows a simple model illustrating how an elapsed time counter is generated and used by a Discrete-Time Integrator block within a triggered subsystem. The following block diagrams show the model `elapseTime_exp`, which contains subsystem `Amplifier`, which includes a Discrete-Time Integrator block.



elapseTime_exp Model



Amplifier Subsystem

A 32-bit timer for the base rate (the only rate in this model) is defined within the `rtModel` structure, as follows, in *`model`*`.h`.

```
-/*
 * Timing:
 * The following substructure contains information regarding
 * the timing information for the model.
 */
  struct {
    time_T stepSize;
    uint32_T clockTick0;
    uint32_T clockTickH0;
    time_T stepSize0;
    time_T tStart;
    time_T tFinal;
    time_T timeOfLastOutput;
    void *timingData;
    real_T *varNextHitTimesList;
    SimTimeStep simTimeStep;
    boolean_T stopRequestedFlag;
    time_T *sampleTimes;
    time_T *offsetTimes;
    int_T *sampleTimeTaskIDPtr;
    int_T *sampleHits;
    int_T *perTaskSampleHits;
    time_T *t;
    time_T sampleTimesArray[1];
    time_T offsetTimesArray[1];
    int_T sampleTimeTaskIDArray[1];
    int_T sampleHitArray[1];
    int_T perTaskSampleHitsArray[1];
    time_T tArray[1];
  } Timing;
```

Had the target been ERT instead of GRT, the `Timing` structure would have been pruned to contain only the data required by the model, as follows:

```
/* Real-time Model Data Structure */ (for ERT!)
struct _RT_MODEL_elapseTime_exp_Tag {

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

```
};
```

Storage for the previous-time value of the `Amplifier` subsystem (`Amplifier_PREV_T`) is allocated in the `D_Work(states)` structure in *`model`*`.h`.

```
typedef struct D_Work_elapseTime_exp_tag {
  real_T DiscreteTimeIntegrator_DSTATE; /* '<S1>/Discrete-Time
                                           Integrator' */
  int32_T clockTickCounter;        /* '<Root>/Pulse Generator' */
  uint32_T Amplifier_PREV_T;       /* '<Root>/Amplifier' */
} D_Work_elapseTime_exp;
```

These structures are declared in *`model`*`.c`:

```
/* Block states (auto storage) */
D_Work_elapseTime_exp elapseTime_exp_DWork;
.
.
.
/* Real-time model */
rtModel_elapseTime_exp elapseTime_exp_M_;
rtModel_elapseTime_exp *elapseTime_exp_M = &elapseTime_exp_M_;
```

The elapsed time computation is performed as follows within the *`model`*`_step` function:

```
/* Output and update for trigger system: '<Root>/Amplifier' */
   uint32_T rt_currentTime =
     ((uint32_T)elapseTime_exp_M->Timing.clockTick0);
   uint32_T rt_elapseTime = rt_currentTime -
     elapseTime_exp_DWork.Amplifier_PREV_T;
   elapseTime_exp_DWork.Amplifier_PREV_T = rt_currentTime;
```

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 = elapseTime_exp_DWork.DiscreteTimeIntegrator_DSTATE;

/* Update for DiscreteIntegrator: '<S1>/Discrete-Time Integrator'*/
   elapseTime_exp_DWork.DiscreteTimeIntegrator_DSTATE += 0.3 *
   (real_T)rt_elapseTime * 1.5 ;
```

Because the triggered subsystem maintains the elapsed time, the TLC implementation of the Discrete-Time Integrator block needs only a single call to `LibGetElapseTime` to access the elapsed time value.

Limitations on the Use of Absolute Time

- "About Absolute Time Limitations" on page 1-78
- "Logging Absolute Time" on page 1-78
- "Absolute Time in Stateflow Charts" on page 1-78
- "Blocks that Depend on Absolute Time" on page 1-78

About 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 1-69 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 1-70 for more information.

Logging Absolute Time

If you log time values by opening the Configuration Parameters dialog box and enabling **Data Import/Export** > **Save to workspace** > **Time**, your model uses absolute time.

Absolute Time in Stateflow Charts

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.

Blocks that Depend on Absolute 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.

Configure Scheduling

- “Configure Start and Stop Times” on page 1-79
- “Configure the Solver Type” on page 1-80
- “Configure the Tasking Mode” on page 1-80

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 Wind River Tornado (VxWorks 5.x) targets, you can override the stop time when running a generated program from the Microsoft Windows command prompt or UNIX[1] 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 "Limitations on the Use of Absolute Time" on page 1-78 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 18-8 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.

Configure the Tasking Mode

The Simulink Coder product supports both single-tasking and multitasking modes for periodic sample times. See "Scheduling" on page 1-4 for details.

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

Supported Products and Block Usage

In this section...

"Related Products" on page 1-81
"Simulink Built-In Blocks That Support Code Generation" on page 1-83
"Simulink Block Data Type Support Table" on page 1-101
"Block Set Support for Code Generation" on page 1-101

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
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
Gauges Blockset™	Linking generated code executing on a target system with graphical instrumentation in a Simulink model
Model-Based Calibration Toolbox™	Developing processes for systematically identifying optimal balance of engine performance, emissions, and fuel economy, and reusing statistical models for control

Product	Extends Code Generation Capabilities for ...
	design, hardware-in-the-loop (HIL) testing, or powertrain simulation
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
Real-Time Windows Target™	Rapid prototyping or hardware-in-the-loop (HIL) simulation of control system and signal processing algorithms
SimDriveline™	Driveline (drivetrain) systems
SimElectronics®	Electronic and electromechanical systems
SimHydraulics®	Hydraulic power and control systems
SimMechanics™	Three-dimensional mechanical systems
SimPowerSystems™	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: • Nonlinear IDNLGREY Model, IDDATA Source, IDDATA Sink, and estimator blocks • Nonlinear ARX models that contain custom regressors • `neuralnet` nonlinearities • `customnet` 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 Simulink Coder and Embedded Coder 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.

- 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

<table>
<tr><th>Block</th><th>Support Notes</th></tr>
<tr><td>Fixed-Point State-Space</td><td>The Simulink Coder software 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.</td></tr>
<tr><td>Transfer Fcn Direct Form II</td><td rowspan="14">• The Simulink Coder software 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.
• Generated code might rely on <code>memcpy</code> or <code>memset</code> (<code>string.h</code>).</td></tr>
<tr><td>Transfer Fcn Direct Form II Time Varying</td></tr>
<tr><td>Unit Delay Enabled</td></tr>
<tr><td>Unit Delay Enabled External IC</td></tr>
<tr><td>Unit Delay Enabled Resettable</td></tr>
<tr><td>Unit Delay Enabled Resettable External IC</td></tr>
<tr><td>Unit Delay External IC</td></tr>
<tr><td>Unit Delay Resettable</td></tr>
<tr><td>Unit Delay Resettable External IC</td></tr>
<tr><td>Unit Delay With Preview Enabled</td></tr>
<tr><td>Unit Delay With Preview Enabled Resettable</td></tr>
<tr><td>Unit Delay With Preview Enabled Resettable External RV</td></tr>
<tr><td>Unit Delay With Preview Resettable</td></tr>
<tr><td>Unit Delay With Preview Resettable External RV</td></tr>
</table>

Additional Math and Discrete: Increment/Decrement

Block	Support Notes
Decrement Real World	The Simulink Coder software 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 Simulink Coder software 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. 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.
Integrator	
"Integrator Limited"	
PID Controller	
PID Controller (2DOF)	
"Second-Order Integrator"	
"Second-Order Integrator Limited"	
State-Space	
Transfer Fcn	
Transport Delay	
"Variable Time Delay"	
Variable Transport Delay	
Zero-Pole	

Discontinuities

Block	Support Notes
Backlash	Supports code generation.
"Coulomb and Viscous Friction"	The Simulink Coder software 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 Simulink Coder software 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 Simulink Coder software 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.
Relay	Support code generation.
Saturation	

Block	Support Notes
Saturation Dynamic	The Simulink Coder software 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	• The Simulink Coder software 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	• Generated code might rely on `memcpy` or `memset` (`string.h`). • Depends on absolute time when used inside a triggered subsystem hierarchy.
Discrete Filter	Support code generation.
Discrete FIR Filter	
PID Controller	• Generated code might rely on `memcpy` or `memset` (`string.h`). • Depends on absolute time when used inside a triggered subsystem hierarchy.
PID Controller (2DOF)	
Discrete State-Space	Generated code might rely on `memcpy` or `memset` (`string.h`).

Block	Support Notes
Discrete Transfer Fcn	
Discrete Zero-Pole	
Discrete-Time Integrator	Depends on absolute time when used inside a triggered subsystem hierarchy.
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.
Tapped Delay	
Transfer Fcn First Order	The Simulink Coder software 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	Generated code might rely on `memcpy` or `memset` (`string.h`).
Zero-Order Hold	Supports code generation.

Logic and Bit Operations

Block	Support Notes
Bit Clear	Support code generation.
Bit Set	
Bitwise Operator	
Combinatorial Logic	
Compare to Constant	
Compare to Zero	

Block	Support Notes
Detect Change	Generated code might rely on `memcpy` or `memset` (`string.h`).
Detect Decrease	
Detect Fall Negative	
Detect Fall Nonpositive	
Detect Increase	
Detect Rise Nonnegative	
Detect Rise Positive	
Extract Bits	Support code generation.
Interval Test	
Interval Test Dynamic	
Logical Operator	
Relational Operator	
Shift Arithmetic	

Lookup Tables

Block	Support Notes
Cosine	The Simulink Coder software 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.
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	

Block	Support Notes
“Sine”	The Simulink Coder software 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	
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)	

Block	Support Notes
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	
Rounding Function	
Sign	
“Signed Sqrt”	
Sine Wave Function	• 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	

Block	Support Notes
Subtract	
Sum	
Sum of Elements	
Trigonometric Function	Functions `asinh`, `acosh`, and `atanh` 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 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
Check Static Gap	

Block	Support Notes
Check Static Lower Bound	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 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	
For Each Subsystem	
For Iterator Subsystem	
Function-Call Generator	

Block	Support Notes
Function-Call Split	
Function-Call Subsystem	
If	
If Action Subsystem	
Model	
Subsystem	
Switch Case	
Switch Case Action Subsystem	
Triggered 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.

Block	Support Notes
Rate Transition	• Generated code might rely on `memcpy` or `memset` (`string.h`). • Cannot use inside a triggered subsystem hierarchy.
Signal Conversion	Support code generation.
Signal Specification	
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.

Block	Support Notes
	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.
Merge	When multiple signals connected to a Merge block have 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 all signals connected to Merge blocks in the group.
Multiport Switch	Support code generation.
Mux	
Selector	
Switch	Generated code might rely on `memcpy` or `memset` (`string.h`).

Sinks

Block	Support Notes
Display	Ignored for code generation.
“Floating Scope”	
“Outport” (Out1)	Supports code generation.
“Scope”	Ignored for code generation.
Stop Simulation	• 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.

Block	Support Notes
Terminator	Supports code generation.
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	• The Simulink Coder software 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	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.
Enumerated Constant	Supports code generation.
From File	Ignored for code generation.
From Workspace	
Ground	Support code generation.
“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

Block	Support Notes
	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	• 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	• The Simulink Coder software 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 Simulink Coder software 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.
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
Signal Generator	

Block	Support Notes
	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.
Sine Wave	• 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.

User-Defined

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

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 1-102
"Sample Time Propagation" on page 1-104
"Latches for Subsystem Blocks" on page 1-105
"Block Execution Order" on page 1-105
"Algebraic Loops" on page 1-106

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

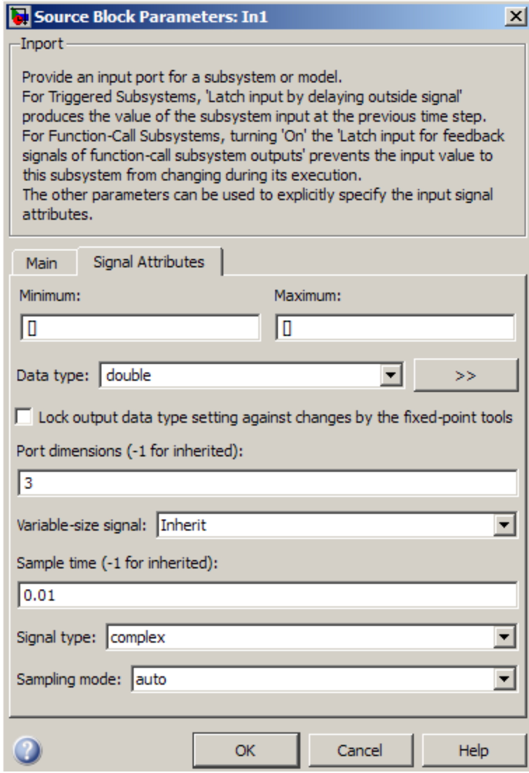
- Data type
- Line widths
- Sample times

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

```
showblockdatatypetable
```

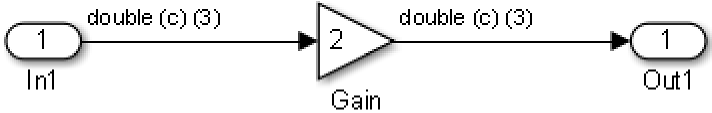
at the MATLAB prompt, or (from the Help browser) clicking the command above.

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.

Constant Block Sample Times

You can specify a sample time for Constant blocks. This has certain implications for code generation.

When a sample time of `inf` is selected for a Constant block,

- If **Inline parameters** is on, the block takes on a constant sample time, and propagates a constant sample time downstream.
- If **Inline parameters** is off, the Constant block inherits its sample time – which is nonconstant – and propagates that sample time downstream.

Generated code for a block differs when it has a constant sample time; its outputs are represented in the constant block outputs structure instead of in the general block outputs structure. The generated code thus reflects that the Constant block propagates

a constant sample time downstream if a sample time of `inf` is specified and **Inline parameters** is on.

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...	You Can 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 Simulink Coder code generator

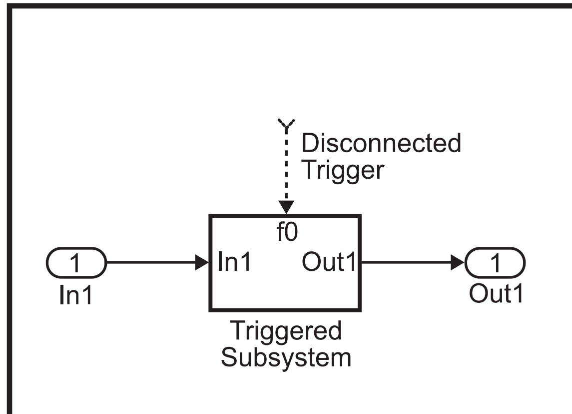
- Preserves latches in generated code regardless of optimizations that might be set
- Places the code for latches at the start of a subsystem's output/update function

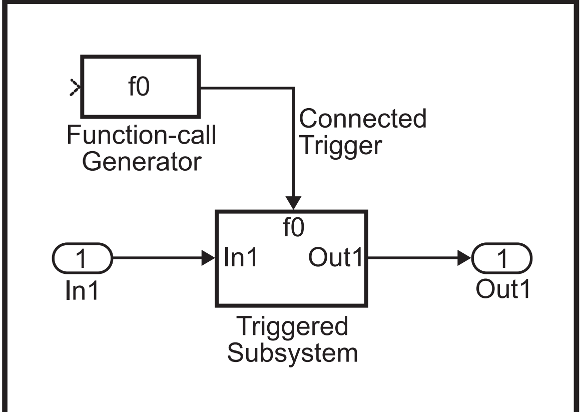
For more information on these options, see the description of the Inport block in the Simulink documentation.

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 Simulink Coder product is to build individual models separately and then manually code the I/O between the models. This approach places the burden of data consistency between models on the developer of the models. Another approach is to let the Simulink and Simulink Coder products maintain data consistency between rates and generate multirate code for use in a multitasking environment. The Simulink Rate Transition block is able to interface both periodic and asynchronous signals. For a description of the Simulink Coder libraries, see “Handle Asynchronous Events” on page 1-31. For more information on multirate code generation, see “Scheduling” on page 1-4.

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

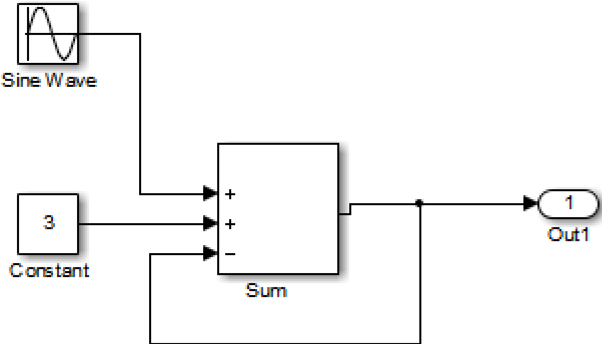
- `x = y + 2`
- `y = -x`

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

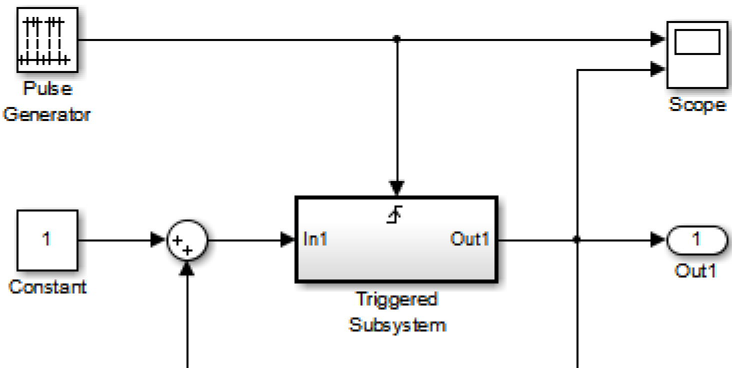
An algebraic loop exists whenever the output of a block having direct feedthrough (such as Gain, Sum, Product, and Transfer Fcn) is fed back as an input to the same block. The Simulink engine is often able to solve models that contain algebraic loops, such as the next diagram.



The Simulink Coder software 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 Simulink Coder 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 Simulink Coder software 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.

Subsystems

- "Code Generation of Subsystems" on page 2-2
- "Generate Code and Executables for Individual Subsystem" on page 2-4
- "Inline Subsystem Code" on page 2-7
- "Generate Subsystem Code as Separate Function and Files" on page 2-10
- "Generate Reusable Function for Identical Subsystems Within a Model" on page 2-11
- "Optimize Code for Identical Nested Subsystems" on page 2-14
- "Generate Reusable Code for Subsystems Containing S-Function Blocks" on page 2-15
- "Generate Reusable Code from Stateflow Charts" on page 2-16
- "Code Reuse Limitations for Subsystems" on page 2-17
- "Code Reuse For Subsystems Shared Across Models" on page 2-19
- "Reusable Library Subsystem" on page 2-20
- "Code Generation of Constant Parameters" on page 2-22
- "Shared Constant Parameters for Code Reuse" on page 2-23
- "Generate Reusable Code for Subsystems Shared Across Models" on page 2-27
- "Determine Why Subsystem Code Is Not Reused" on page 2-35

Code Generation of Subsystems

For you to control how code is generated for a nonvirtual subsystem, the Simulink Coder software 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 run the sl_subsys_semantics model.

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 2-7
Generate code for only a subsystem.	“Generate Code and Executables for Individual Subsystem” on page 2-4
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 2-10
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 2-11

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 2-27 and "Code Reuse For Subsystems Shared Across Models" on page 2-19

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 Simulink Coder software 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".

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 2-19.

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.

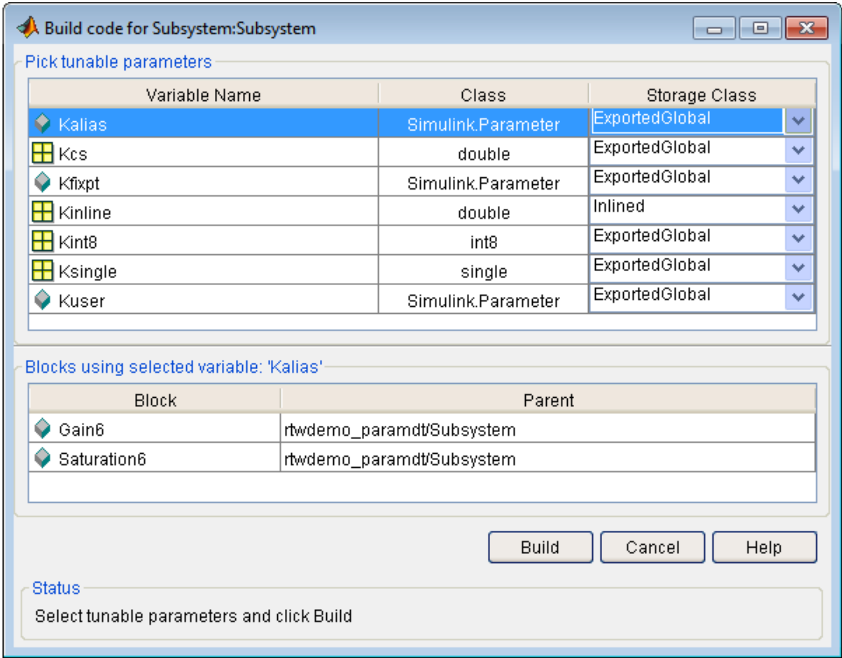
   Alternatively, in the current model, click a subsystem and then from the **Code** menu, select **C/C++ Code > Build Selected Subsystem**.

   Note When you select **Build This Subsystem**, if the model is operating in external mode, the Simulink Coder build process automatically turns off external mode for the duration of the build. Simulink Coder then 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 "Parameters".

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 can use the right-click subsystem build. For more information on S-functions, see "Automate S-Function Generation".

Subsystem Build Limitations

The following limitations apply to building subsystems:

- When you right-click 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`.

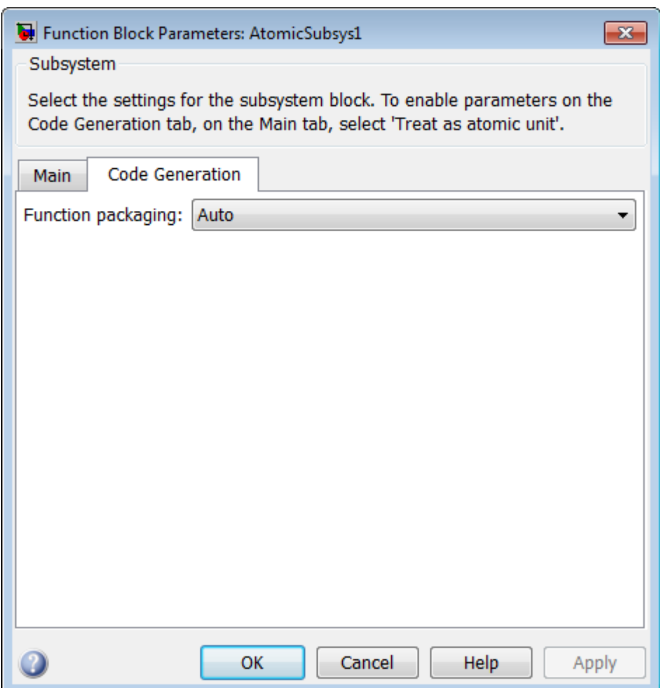
To inline subsystem code, select `Inline`. The `Inline` option explicitly directs the code generator to inline the subsystem unconditionally.

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 VxWorks block library (`vxlib1`) shipped with the Simulink Coder product and use the `SS_OPTION_FORCE_NONINLINED_FCNCALL` option.[2]

2. 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 2-11 or "Generate Reusable Code for Subsystems Shared Across Models" on page 2-27.
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.

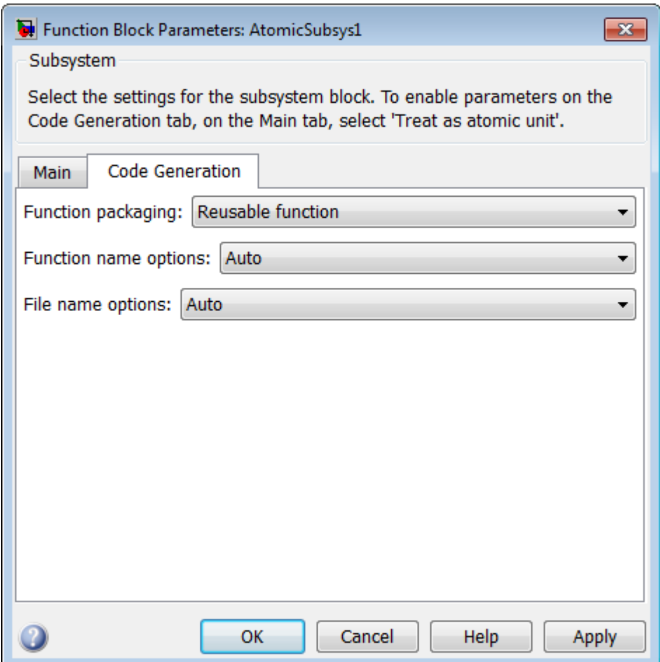
To determine reusability of the subsystem code, the code generator performs a checksum to determine if subsystems are identical. The generated function has arguments, for example, for block inputs and outputs (`rtB_*`), continuous states (`rtDW_*`), parameters (`rtP_*`).

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 2-13.

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 2-17.

Considerations for Function Packaging Options `Auto` and `Reusable function`

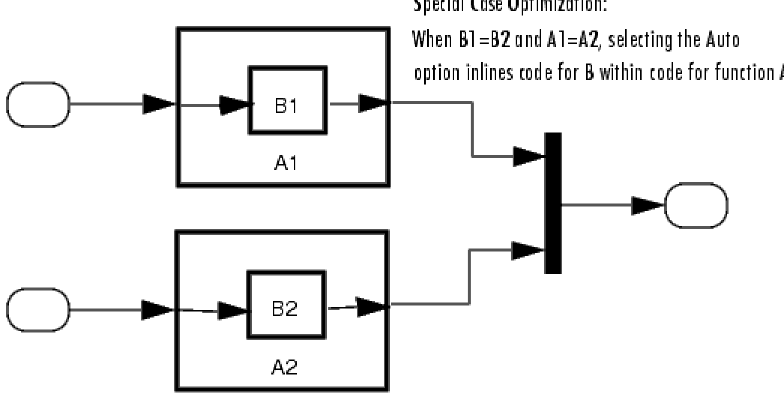
When you want multiple instances of a subsystem to be represented as one reusable function, you can designate each one of them as `Auto` or as `Reusable function`. Use one or the other, because using both creates two reusable functions, one for each specification. The outcomes of these choices differ 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 inline, then the generated code is separate functions without arguments for each subsystem instance.
- `Reusable function` yields a separate function with arguments for each subsystem instance in the model.

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".

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 2-11.

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:

- 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.

If you select `Reusable function`, and the code generator determines that code for a subsystem cannot be reused, it generates a separate function that is not reused. The code generation report might show that the separate function is reusable, even if it is used by only one subsystem. If you prefer that subsystem code be inlined in such circumstances rather than deployed as functions, then 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”)
- 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 reusable library subsystem code is placed in the shared utilities folder and 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 might 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`.

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 2-20.

For an example, see “Generate Reusable Code for Subsystems Shared Across Models” on page 2-27.

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 2-17.

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 2-27.

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 string 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_mgdjlngd`
- File name: `RLS_mgdjlnd.c` and `RLS_mgdjlnd.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 Outport

If a reusable library subsystem is connected to the root outport, reuse does not happen with identical subsystems that are not connected to the root outport. 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 2-23.

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 string:

```
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 2-22.

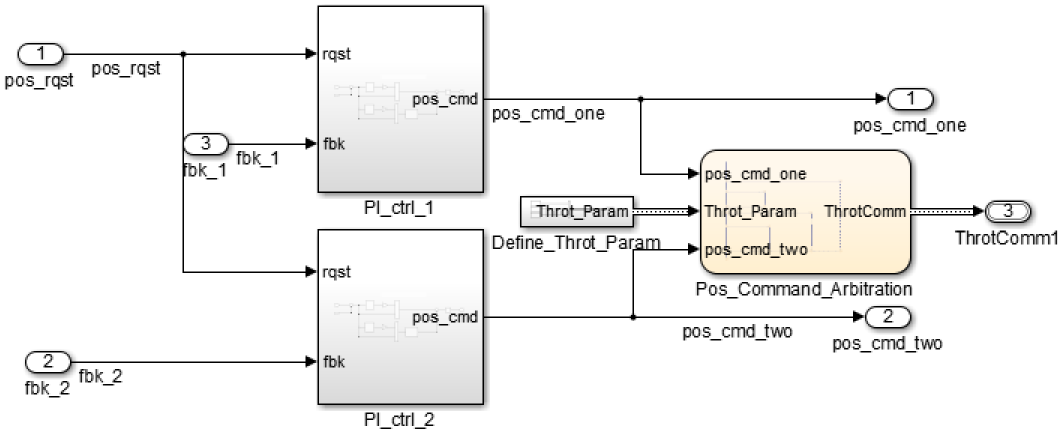
Suppress Shared Constants in the Generated Code

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_Ygnal0wM3c14[7];
const real_T rtCP_pooled_Ygnal0wM3c14[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:

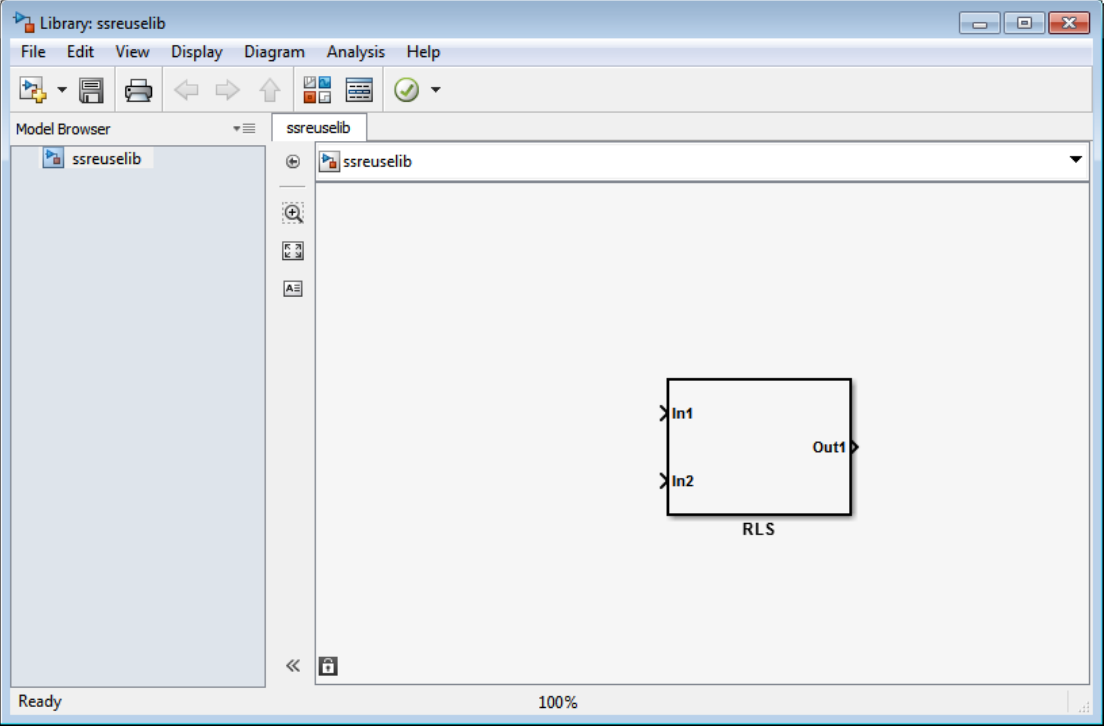
- A constant is referenced by a non-inlined S-function.
- A constant has a user-defined type where **Data Scope** is not set to `Exported`.

Generate Reusable Code for Subsystems Shared Across Models

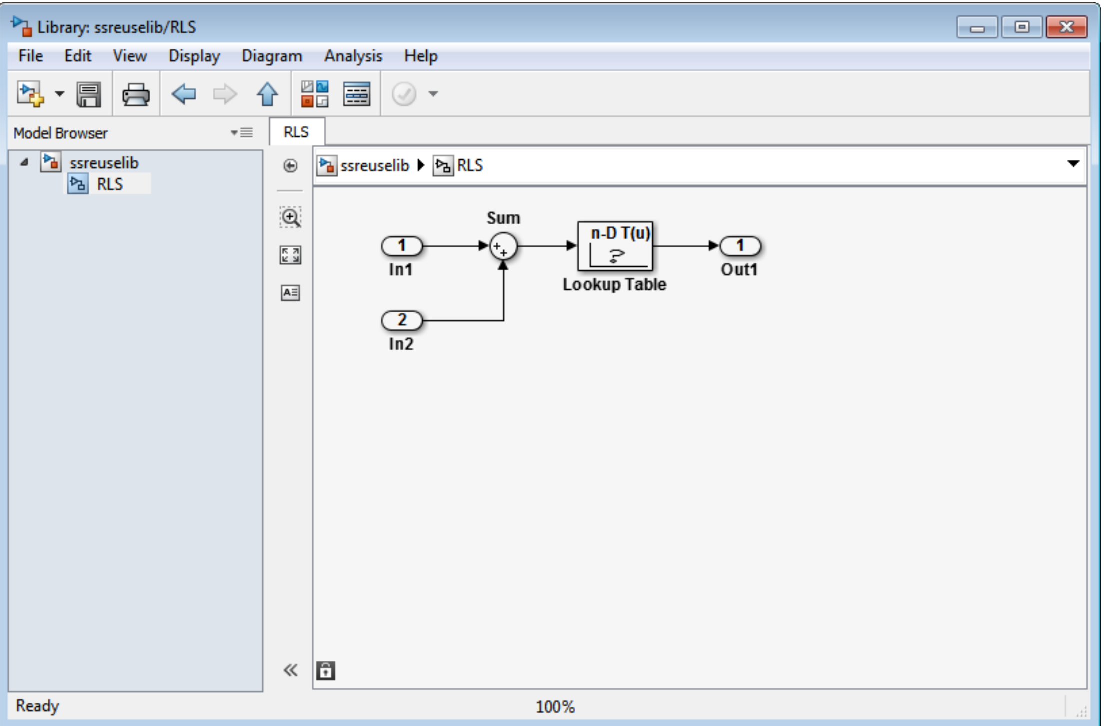
This example shows how to configure a reusable library subsystem and generate a reusable function for a subsystem shared across referenced models. The result is reusable code for the subsystem, which is generated to the shared utility folder (`slprj/`*`target`*`/_sharedutils`).

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:

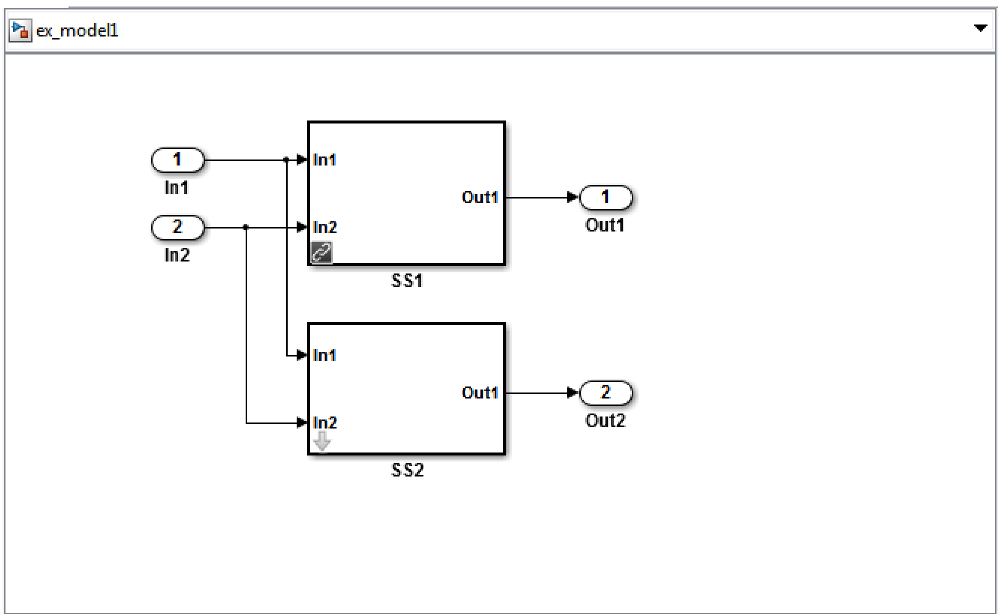
- Select **Treat as an atomic unit**.
- On the **Code Generation** tab:
  - Set **Function packaging** to `Reusable function`.
  - Set **Function name options** and **File name options** to `Auto`.

4 Click **Apply** and **OK**.

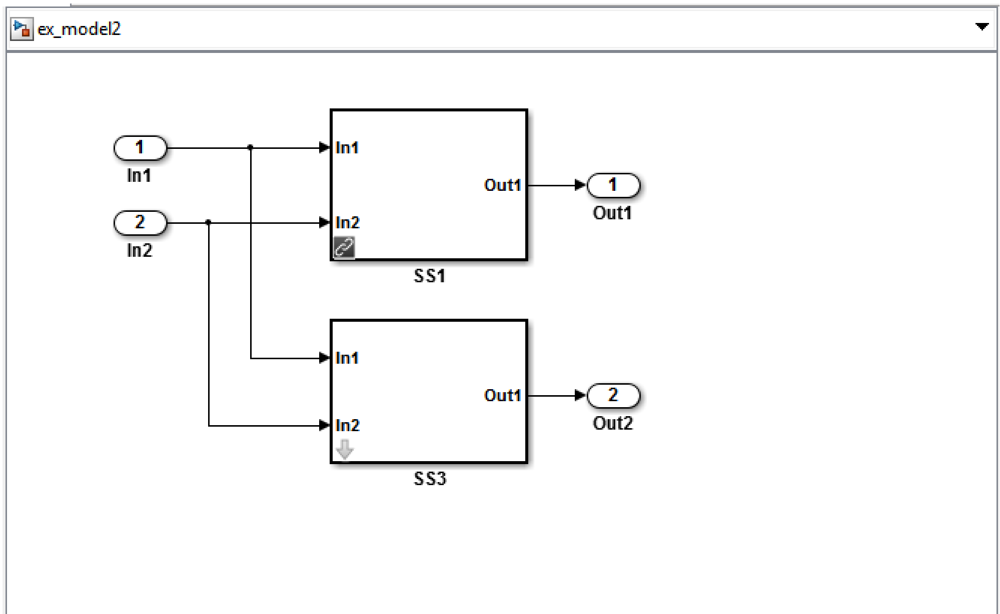
5 Save the reusable library subsystem as `ssreuselib`, which creates a file, `ssreuselib.slx`.

Create the example model.

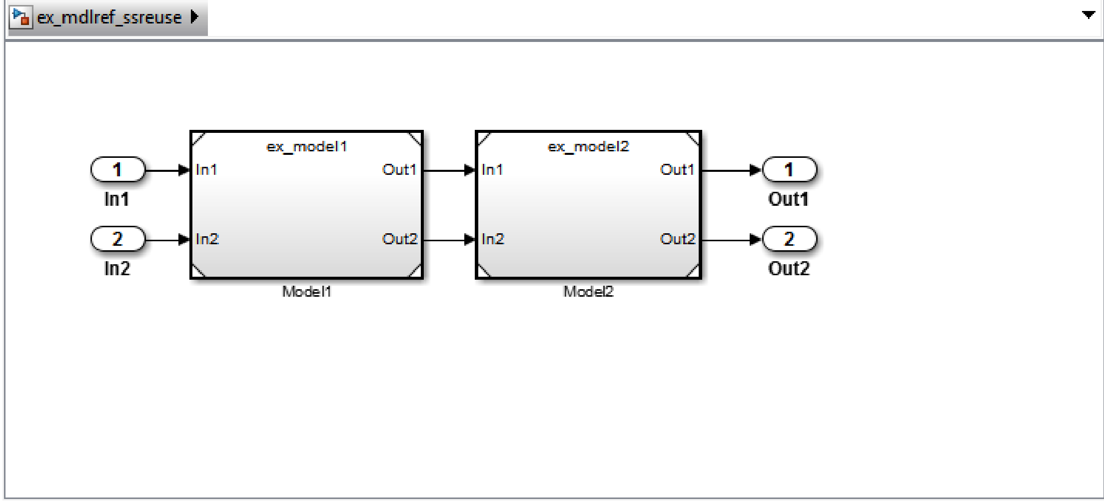
1. Create a model which includes one instance of RLS from ssreuselib. Name this subsystem SS1. Add another subsystem and name it SS2. Name the model ex_model1.



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_mdlref_ssreuse`.



Set configuration parameters of the top model.

1. With model `ex_mdlref_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:
   - Select **Inline parameters**.
   - 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_mdlref_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 string to name the reused function. The code for subsystem SS1 is in RLS_9NCnbXn8.c and RLS_9NCnbXn8.h.

Code Generation Report

Find: Match Case

Contents

Summary

Subsystem Report

Traceability Report

Static Code Metrics Report

Code Replacements Report

Generated Code

[-] **Model files**

ex_model1.c

ex_model1.h

ex_model1_private.h

ex_model1_types.h

[-] **Shared files**

RLS_9NCnbXn8.c

RLS_9NCnbXn8.h

look1_binlxpw.c

look1_binlxpw.h

model_reference_types.h

rtwtypes.h

File: RLS_9NCnbXn8.c

```
1  /*
2   * File: RLS_9NCnbXn8.c
3   *
4   * Code generated for Simulink model 'ex_model1'.
5   *
6   * Model version                  : 1.2
7   * Simulink Coder version         : 8.7 (R2014b Prerelease) 20-Jun-2014
8   * C/C++ source code generated on : Tue Jul 01 11:03:37 2014
9   */
10
11 #include "RLS_9NCnbXn8.h"
12
13 /* Output and update for atomic system: 'RLS' ('ssreuselib:1') */
14 void RLS_9NCnbXn8(real_T rtu_In1, real_T rtu_In2, real_T *rty_Out1, const real_T
15                   rtp_y[11], const real_T rtp_x[11])
16 {
17   /* Lookup_n-D: 'Lookup Table' ('ssreuselib:4') incorporates:
18    *  Sum: 'Sum' ('ssreuselib:5')
19    */
20   *rty_Out1 = look1_binlxpw(rtu_In1 + rtu_In2, rtp_x, rtp_y, 10U);
21 }
22
23 /*
24  * File trailer for generated code.
25  *
26  * [EOF]
27  */
```

OK Help

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 2-35
- “Share a Configuration Across Referenced Models”
- “Generate Reusable Function for Identical Subsystems Within a Model” on page 2-11
- “Save a Referenced Configuration Set”

More About

- “Code Generation of Subsystems” on page 2-2
- “Code Reuse For Subsystems Shared Across Models” on page 2-19
- “Code Reuse Limitations for Subsystems” on page 2-17
- “Libraries”

Determine Why Subsystem Code Is Not Reused

Due to the limitations described in “Code Reuse Limitations for Subsystems” on page 2-17, 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 2-35. If you cannot determine why based on the report, see “Compare Subsystem Checksum Data” on page 2-35.

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”). The Subsystems section contains:

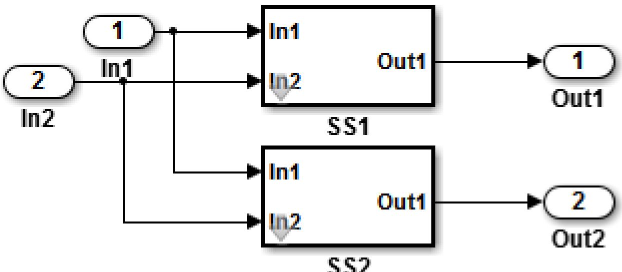
- A table that summarizes how nonvirtual subsystems were converted to generated code.
- Diagnostic information that describes why the contents of some subsystems were not generated as reusable code.

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 2-17. 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(['String 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
```

11 Run the script. The following example assumes that you named the script `check_details`.

```
check_details
String 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.

3

Code Generation of Functions and Function Callers

Modeling Functions and Callers for Code Generation

In this section...

"Functions and Callers" on page 3-2

"Input and Output Arguments" on page 3-2

"Function and Function Caller Definitions Across Models" on page 3-3

"Code Generation Files" on page 3-3

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 "Functions and Function Callers", "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 is a GRT target.
- The system target is an ERT target and you do not have an Embedded Coder license.

- The system target is an ERT target and you have an Embedded Coder license. In the Configuration Parameters dialog box, on the **Code Generation > Code Placement** pane, from the **File packaging format** drop-down list, you select `Modular`.

For a model named `model` and a function named `fn1`, the code generator creates the following files.

Files with GRT System Target or ERT Target with Modular File Packaging

File	Folder	Contents
`fn1.c`	`model_`*`target`*`_rtw`	The code for the function.
`fn1_private.h`	`model_`*`target`*`_rtw`	This header file includes declarations and header files for the function, including `fn1.h`.
`fn1.h`	`slprj/`*`target`*`/_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.
`model.c`	`model_`*`target`*`_rtw`	Calls to the function.

If you have an Embedded Coder license, an ERT target, and **File packaging format** is set to `Compact` or `Compact (with separate data file)`, the code generator creates the following files.

Generated Code Files With ERT System Target and Compact File Packaging

File	Folder	Contents
`model.c`	`model_ert_rtw`	The code for the function and calls to the function.
`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.

For more information, see "Generate Code for Functions and Callers" on page 3-6.

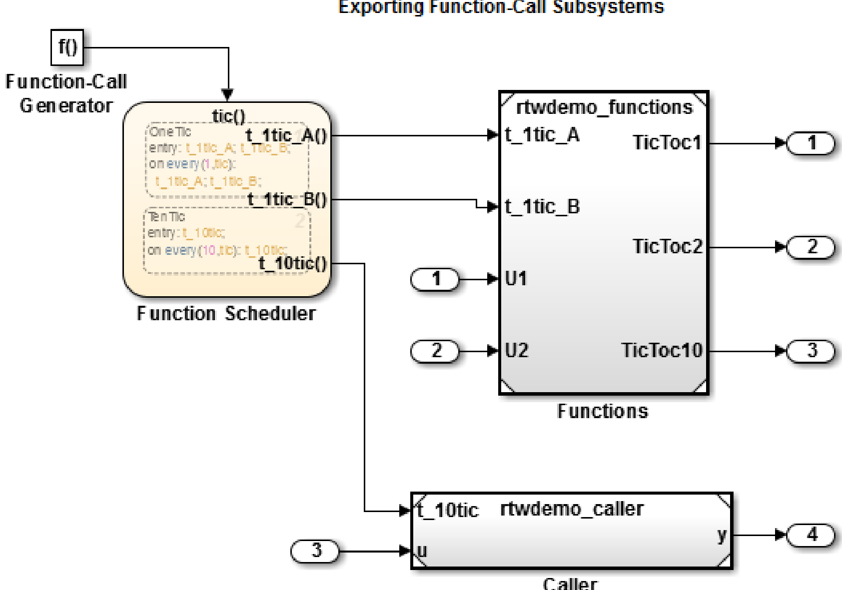
Generate Code for Functions and Callers

In this section...

"Generate Code for the Function Definition" on page 3-6
"Generate Code for the Caller Definition" on page 3-8

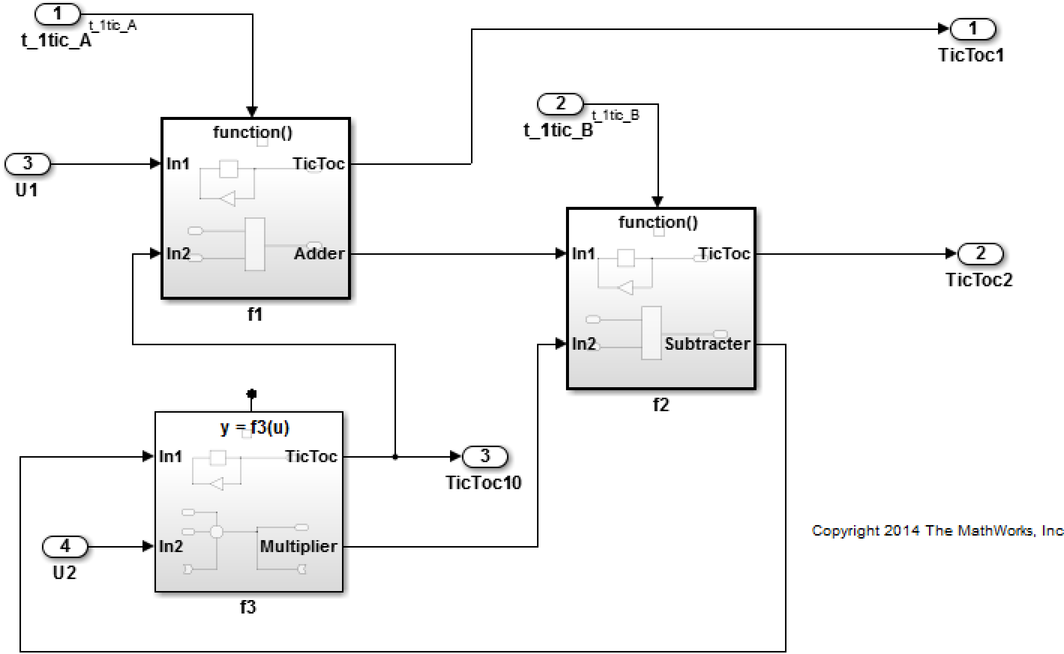
This example shows how the code generator translates Simulink Function blocks and Function Caller blocks into C code.

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)`.



2 In the Configuration Parameters dialog box, on the **Code Generation** pane, click **Generate Code**.

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: '<S3>/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: '<S3>/Delay'
   */
  rtY.TicToc10 = rtDWork.Delay_DSTATE;

  /* Sum: '<S3>/Sum' incorporates:
   *  Inport: '<Root>/U2'
   *  SignalConversion: '<S3>/TmpLatchAtIn1Outport1'
   *  SignalConversion: '<S3>/TmpSignal ConversionAtuOutport1'
   */
  rtB.Sum = (rtB.Subtract + rtU.U2) + rtu_u;

  /* SignalConversion: '<S3>/TmpSignal ConversionAtyInport1' */
  *rty_y = rtB.Sum;

  /* Update for UnitDelay: '<S3>/Delay' incorporates:
   *  Gain: '<S3>/Gain'
   */
  rtDWork.Delay_DSTATE = (int8_T)-rtY.TicToc10;
}
```

- 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. On the **Code Generation** pane, click **Generate Code**.

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);
}
```

For more information, see “Modeling Functions and Callers for Code Generation” on page 3-2 and, in the Embedded Coder documentation, for AUTOSAR, “Configure AUTOSAR Client-Server Communication”.

Referenced Models

- "Code Generation for Referenced Models" on page 4-2
- "Generate Code for Referenced Models" on page 4-4
- "Code Generation Folder Structure for Model Reference Targets" on page 4-15
- "Configure Referenced Models" on page 4-16
- "Build Model Reference Targets" on page 4-17
- "Simulink Coder Model Referencing Requirements" on page 4-18
- "Storage Classes for Signals Used with Model Blocks" on page 4-23
- "Inherited Sample Time for Referenced Models" on page 4-26
- "Customize Library File Suffix and File Type" on page 4-28
- "Reusable Code and Referenced Models" on page 4-29
- "Simulink Coder Model Referencing Limitations" on page 4-33

Code Generation for 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 current working folder, and the code for referenced models in a folder named `slprj` within the current working folder. Subfolders in `slprj` provide separate places for different types of files. See “Code Generation Folder Structure for Model Reference Targets” on page 4-15 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 26-78.

Generate Code for Referenced Models

In this section...

"About Generating Code for Referenced Models" on page 4-4
"Create and Configure the Subsystem" on page 4-4
"Convert Model to Use Model Referencing" on page 4-7
"Generate Model Reference Code for a GRT Target" on page 4-11
"Work with Code Generation Folders" on page 4-14

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`).

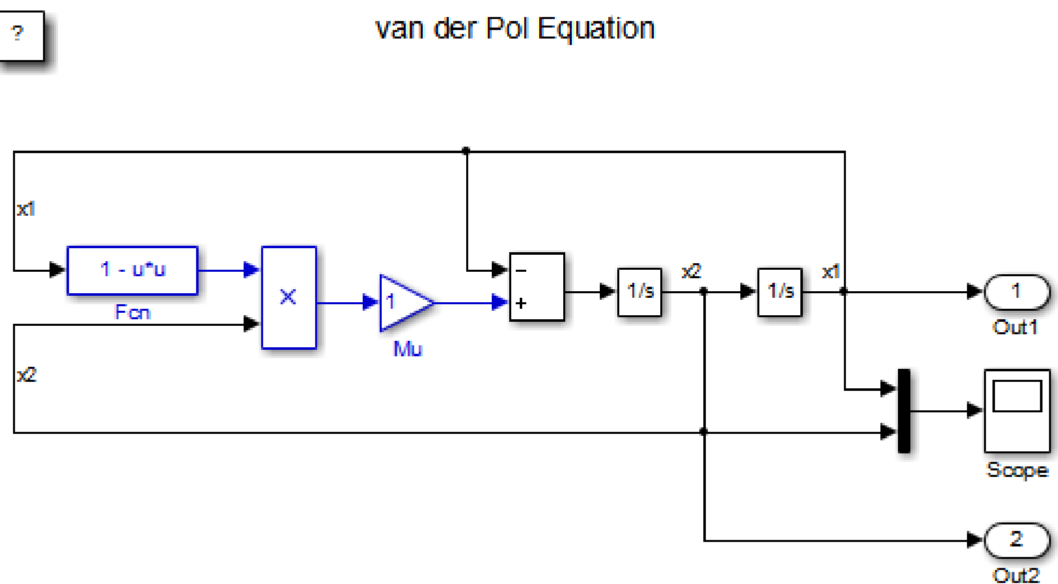
1. In the MATLAB Command Window, create a new working folder wherever you want to work and `cd` into it:

   ```
   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`.

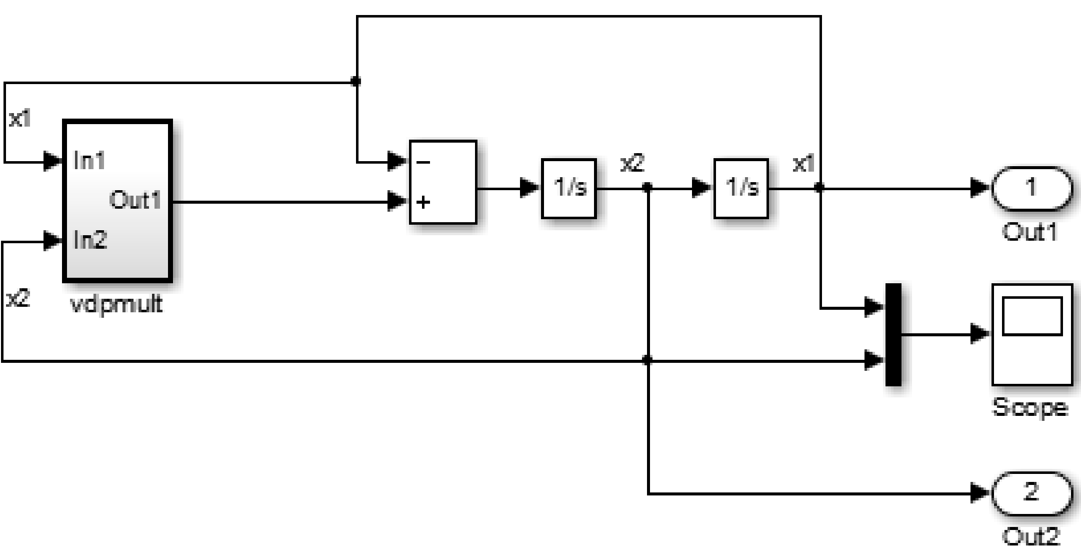
7 Right-click the `vdpmult` block and select **Block Parameters (Subsystem)**.

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:



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`.
   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 `vdp` 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_mdlref_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')
```

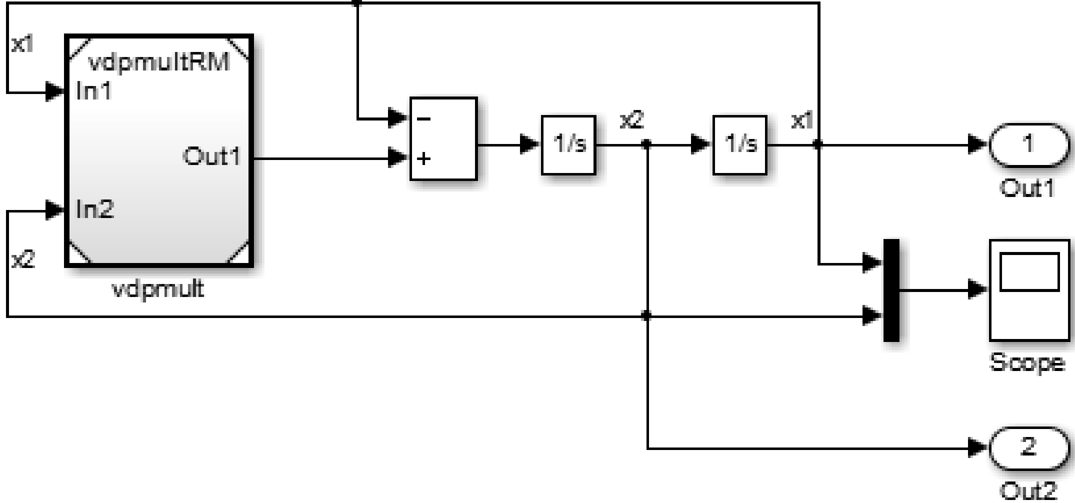
This command:

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
```

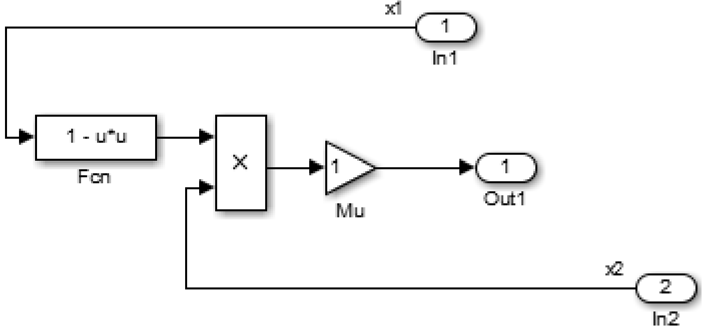
The parent model `vdptop` now looks like this:



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

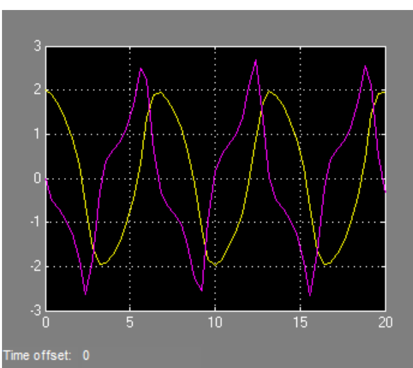
The files in your working folder now consist of the following (not in this order).

File	Description
`vdptop` model file	Top model that contains a Model block where the `vdpmult` subsystem was
`vdpmultRM` model file	Referenced model created for the `vdpmult` subsystem
`vdpmultRM_msf.mexw32`	Static library file (Microsoft Windows platforms only). The file extension is system-dependent and may differ. This file executes when the `vdptop` model calls the Model block `vdpmult`. When called, `vdpmult` in turn calls the referenced model `vdpmultRM`.
`/slprj`	Code generation 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 4-14.

Run the Converted Model

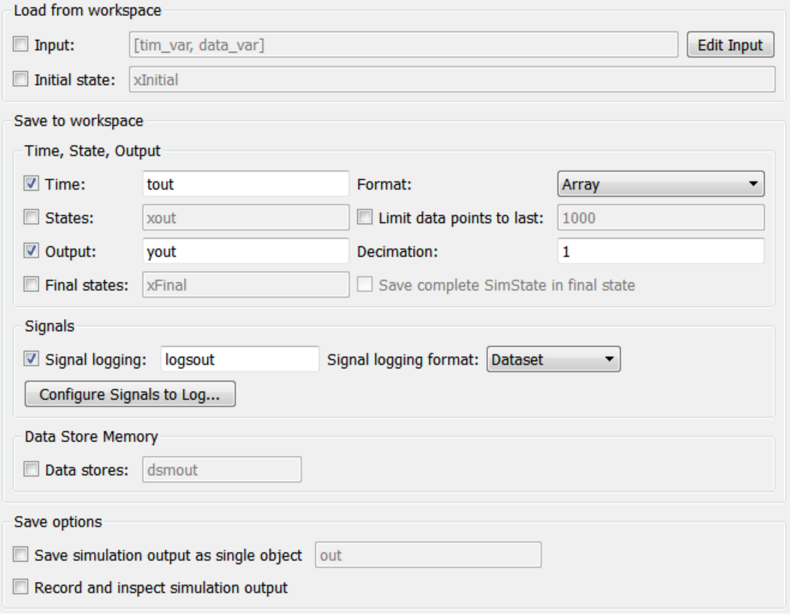
Open the Scope block in `vdptop` if it is not visible. In the `vdptop` window, click the **Run** tool or choose **Run** from the **Simulation** menu. The model calls the `vdpmultRM_msf` simulation target to simulate. The output looks like this:



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 **Save to workspace** section of the right pane, select **Time** and **Output** and *clear* **Data stores**. Click **Apply**. The pane shows the following information:



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 by selecting **Code Generation** in the center pane and then clicking the **Build** button.

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.

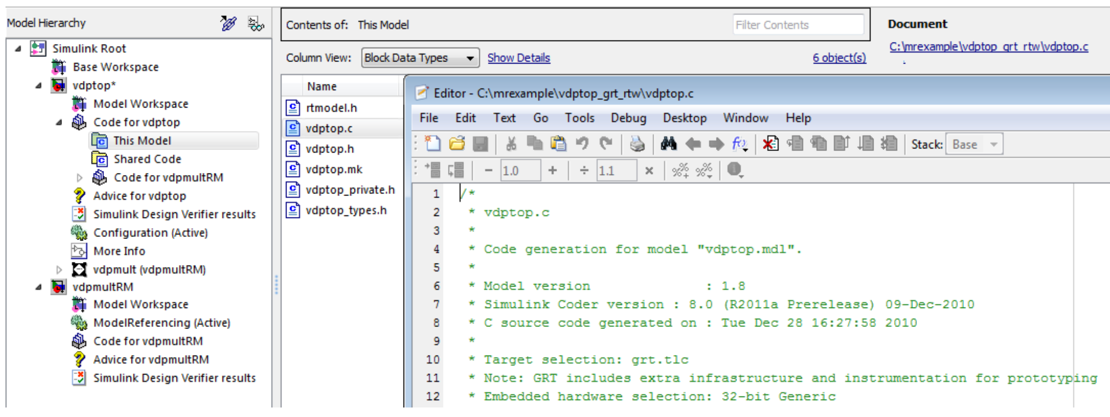
4 Directly under `Code for vdptop`, click `This Model`.

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”.

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 (rooted under the code generation folder `slprj`), like build folders, are created in your current working folder, and this implies certain constraints on when and where model reference targets are built, and 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 for 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.

Code Generation Folder Structure for Model Reference Targets

Code for models referenced by using Model blocks is generated in code generation folders within the current working folder. The top-level code generation folder is named `/slprj`. 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
`slprj/sim/`*`model`*`/`	Simulation target files for referenced models
`slprj/sim/`*`model`*`/tmwinternal`	MAT-files used during code generation
`slprj/`*`target`*`/`*`model`*`/referenced_model_includes`	Header files from models referenced by this model
`slprj/`*`target`*`/`*`model`*	Model reference target files
`slprj/`*`target`*`/`*`model`*`/tmwinternal`	MAT-files used during code generation
`slprj/sl_proj.tmw`	Marker file
`slprj/`*`target`*`/_sharedutils`	Utility functions for model reference targets, shared across models
`slprj/sim/_sharedutils`	Utility functions for simulation targets, shared across models

If you are building code for more than one referenced model within the same working 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” 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 Referencing Pane” 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 4-29 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`. (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 4-33

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. See “Model configuration mismatch” for details.
- 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	**Test hardware** 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.
	Data exchange > Interface	**C API**	The C API check boxes for signals, parameters, and

Dialog Box Pane	Option		Requirement
			states must be the same for top and referenced models.
		ASAP2	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
Code Generation	**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

Dialog Box Pane	Option	Requirement
	Support non-finite numbers	If `off` for top model, must be `off` for referenced models.
	Support complex numbers	If `off` for top model, must be `off` for referenced models.
	Suppress error status in real-time model	If `on` for top model, must be `on` for referenced models.
Data Placement	**Module Naming**	Must be the same for top and referenced models.
	Module Name (if specified)	If set, 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 string. 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 “Code Generation Pane: 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 26-78.

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 Model References” 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 “Inherit Sample Times” 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 string for the suffix and extension. The string 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”. 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.

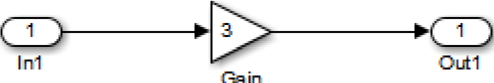
Note You can generate subsystem code using any target configuration available in the System Target File Browser. However, if the S-function target is selected, **Build This Subsystem** and **Build Selected Subsystem** behaves identically to **Generate S-Function**. (See “Automate S-Function Generation” on page 16-21.)

Code Reuse and Model Blocks with Root Inport or Outport Blocks

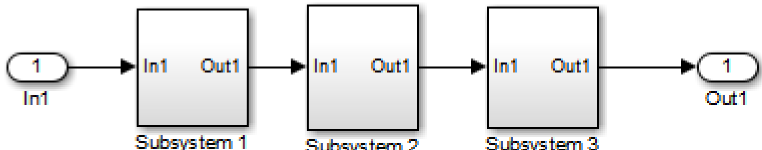
Reusable functions with inputs or outputs connected to a referenced model's root Inport or Outport block can affect code reuse. This means that code for certain atomic subsystems cannot be reused in a model reference context the same way it is reused in a standalone model.

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 **Inline parameters** option enabled 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.

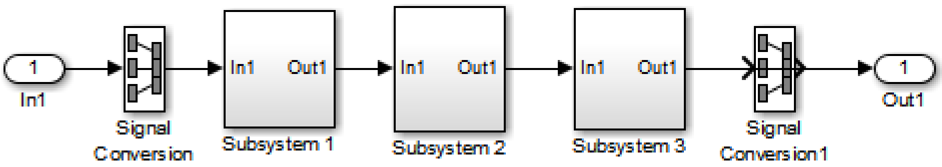
```
void reuse_subsys1_Subsystem1(
                   real_T rtu_0,
                   rtB_reuse_subsys1_Subsystem1 *localB)
{

  /* Gain: '<S1>/Gain' */
  localB->Gain_k = rtu_0 * 3.0;
}
```

When generated as code for a Model block (into an `slprj` 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 Outport 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 4-18.

Customization Limitations

- The code generator ignores custom code settings in the **Configuration Parameter** dialog box and custom code blocks when generating code for a referenced model.
- Referenced models do not support custom storage classes if the parent model has inline parameters off.
- This release does not include Stateflow target custom code in simulation targets generated for referenced models.
- Data type replacement is not supported for simulation target code generation for 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 **Configuration Paramters** > **Data Import/Export** > **Initial state** parameter 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

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 Model Referencing”.

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”.

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.

Combined Models

- "Combined Models" on page 5-2
- "Use GRT with Reusable Function Packaging to Combine Models" on page 5-3

Combined Models

If you want to combine several models (or several instances of the same model) into a single executable, the Simulink Coder product 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 4-4.

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 Embedded Coder target has certain restrictions, relating to embedded processing, that might not be compatible with your application.

The GRT target with the model parameter **Code interface packaging** set to `Reusable function` is a 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 5-3.

To summarize by target, your options are as follows:

Target	Support for Combining Multiple Models?
Generic Real-Time Target (`grt.tlc`)	Yes (using Model blocks or **Code interface packaging** set to `Reusable function`)
Embedded Coder (`ert.tlc`)	Yes
S-function Target (`rtwsfcn.tlc`)	No

Use GRT with Reusable Function Packaging to Combine Models

This section discusses how to use the GRT target, with the model parameter **Code interface packaging** set to `Reusable function`, to combine models into a single program. (See "Combined Models" on page 5-2 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

- “Platform Options for Development and Deployment” on page 6-2
- “Configure Test and Production Target Hardware” on page 6-3
- “Configure Production Hardware Characteristics” on page 6-13
- “Configure Test Hardware Characteristics” on page 6-14
- “Control the Location for Generated Files” on page 6-17
- “Control Generated Files Location Used for Simulation” on page 6-19
- “Control the Location for Code Generation Files” on page 6-21
- “Override Build Folder Settings for Current Session” on page 6-23

Platform Options for Development and Deployment

When you use Simulink software to create and execute a model, and Simulink Coder software to generate code, you may need to 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 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

The same platform might serve in two or possibly three capacities, but they 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 it can do little more than run a downloaded executable file.

When you use Simulink software to execute a model for which you will later generate code, or use Simulink Coder software to generate code for deployment on a *production* target, you must provide information about the production target hardware and the compiler that you will 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 Simulink Coder code generator uses the information to create code that executes with maximum efficiency.

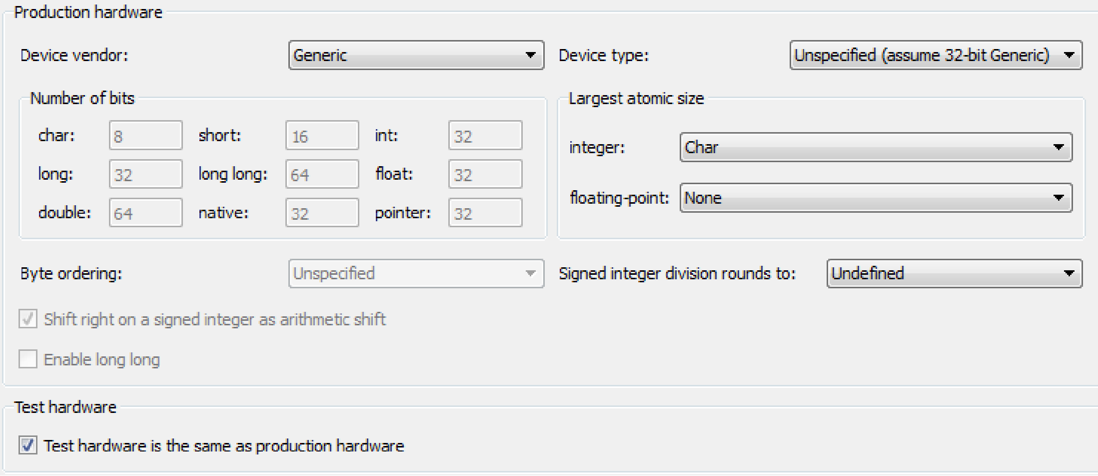
When you generate code for testing on a *test* target, you must additionally provide information about the test target hardware and the compiler that you will use with it. The code generator uses this information to create code that provides bit-true agreement for the results of integer and fixed-point 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 may use very different hardware, and the compilers for the two targets may use different defaults where the C standard does not completely define behavior.

Configure Test and Production Target Hardware

The Configuration Parameters dialog **Hardware Implementation** pane provides parameters that you can use to describe hardware properties, such as data size and byte ordering, and compiler behavior details that may vary with the compiler, such as integer rounding. The **Hardware Implementation** pane contains two subpanes:

- **Production hardware** — Describes the production target hardware and the compiler that you will use with it. This information affects both simulation and code generation.
- **Test hardware** — Describes the test target hardware and the compiler that you will use with it. This information affects only code generation.

The two subpanes provide identical parameters and value choices. By default, the **Hardware Implementation** pane initially looks like this:



The default assumption is that the production hardware and test hardware are the same, so the **Test hardware** subpane by default does not need to specify anything and contains only a checked option labeled **Test hardware is the same as production hardware**. Code generated under this configuration will be suitable for production use, or for testing in an environment identical to the production environment.

If the production hardware and test hardware are not the same, clear the **Test hardware is the same as production hardware** option. The full **Test hardware**

subpane is displayed, initially showing the same values as the **Production hardware** subpane. Configure the **Test hardware** parameters to describe your test hardware. The generated code will be able to execute in the environment specified by the **Test hardware** subpane, but will behave as if it were executing in the environment specified by the **Production hardware** subpane. See "Configure Test Hardware Characteristics" on page 6-14 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.

Parameters with only one possible value cannot be changed. Parameters that have more than one possible value provide a list of legal values. If you specify hardware properties manually, check carefully that their values are consistent with the system target file. Otherwise, the generated code may fail to compile or execute, or may execute but give incorrect results.

Note: Hardware Implementation pane parameters do not control hardware or compiler behavior. They describe hardware and compiler properties to MATLAB software, which uses the information to generate 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 section describes the parameters in the **Production hardware** and **Test hardware** subpanes. Subsequent sections describe considerations that apply only to one or the other subpane. For more about **Hardware Implementation** parameters, see "Hardware Implementation Pane". To see an example of **Hardware Implementation** pane capabilities, run the rtwdemo_targetsettings example.

Identify the Device Vendor

The **Device vendor** parameter gives the name of the device vendor. To set the parameter, select a vendor name from the **Device vendor** menu. Your selection of vendor will determine the available device values in the **Device type** list. If the desired vendor name does not appear in the menu, select `Generic` and then use the **Device type** parameter to further specify the device.

Note:

- 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 6-5.

Identify the Device Type

The **Device type** parameter selects a hardware device among the supported devices listed for your **Device vendor** selection. To set the parameter, select a microprocessor name from the **Device type** menu. If the desired microprocessor does not appear in the menu, change the **Device vendor** to `Generic`.

If you specified the **Device vendor** as `Generic`, examine the listed device descriptions and select the device type that matches your hardware. If no available device type matches, select `Custom`.

If you select a device type for which the target file specifies default hardware properties, the properties appear as initial values in the subpane. Parameters with only one possible value cannot be changed. Parameters that have more than one possible value provide a list of legal values. Select values for your hardware. If the device type is `Custom`, more parameters can be set, and each parameter has a list of possible values.

Register Additional Device Vendor and Device Type Values

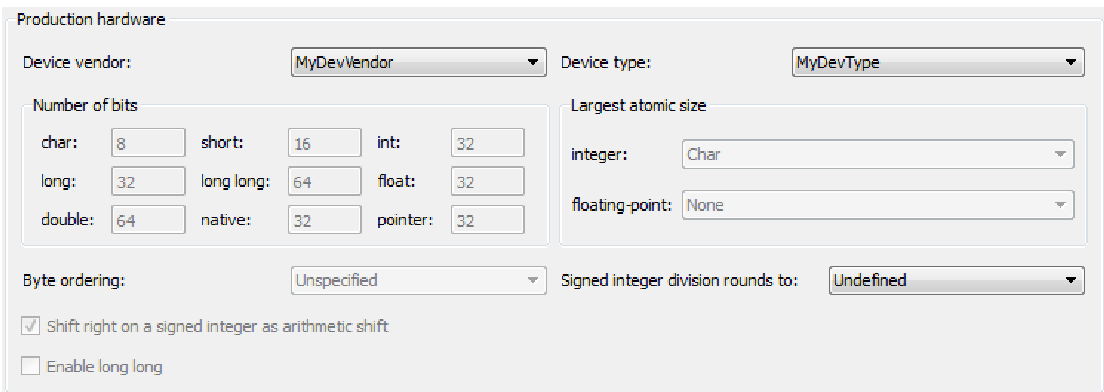
To add **Device vendor** and **Device type** values to the default set that is displayed on the **Hardware Implementation** pane, you can use a hardware device registration API provided by the Simulink Coder software.

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 will be 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
```

If you subsequently select the device in the **Hardware Implementation** pane, it is displayed as follows:



To register multiple devices, you can 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`	String specifying the **Device vendor** value for your hardware device.
`Type`	String specifying the **Device type** value for your hardware device.
`Alias`	Cell array of strings specifying aliases or legacy names that users might specify that should resolve to this device. Specify each alias or legacy name in the format `'Vendor->Type'`. (Embedded Coder software provides the utility functions `RTW.isHWDeviceTypeEq` and `RTW.resolveHWDeviceType` for detecting and resolving alias values or legacy values when testing user-specified values for the target device type.)
`Platform`	Cell array of enumerated string values specifying whether this device should be listed in the **Production hardware** subpane (`{'Prod'}`), the **Test hardware** subpane (`{'Target'}`), or both (`{'Prod', 'Target'}`).
`setWordSizes`	Array of integer sizes to associate with the **Number of bits** parameters **char**, **short**, **int**, **long**, and **native word size**, respectively.
`LargestAtomicInteger`	String specifying an enumerated value for the **Largest atomic size: integer** parameter: `'Char'`, `'Short'`,`'Int'`, or `'Long'`.

Property	Description
`LargestAtomicFloat`	String specifying an enumerated value for the **Largest atomic size: floating-point** parameter: `'Float'`, `'Double'`, or `'None'`.
`Endianess`	String specifying an enumerated value for the **Byte ordering** parameter: `'Unspecified'`, `'Little'` for little Endian, or `'Big'` for big Endian.
`IntDivRoundTo`	String specifying an enumerated value for the **Signed integer division rounds to** parameter: `'Zero'`, `'Floor'`, or `'Undefined'`.
`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 strings specifying which device properties should be enabled (modifiable) in the **Hardware Implementation** pane when this device type is selected. In addition to the `'Endianess'`, `'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.

Simulink Coder 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 the standard Simulink Coder integer type names and maps them to the corresponding Simulink names.

Simulink Coder Integer Type	Simulink Integer Type
boolean_T	boolean
int8_T	int8
uint8_T	uint8
int16_T	int16
uint16_T	uint16
int32_T	int32
uint32_T	uint32

If no ANSI® C type with a matching word size is available, but a larger ANSI C type is available, the Simulink Coder code generator uses the larger type for `int8_T`, `uint8_T`, `int16_T`, `uint16_T`, `int32_T`, and `uint32_T`.

An application can use integer data of length from 1 (unsigned) or 2 (signed) bits up 32 bits. If the integer length matches the length of an available type, the Simulink Coder 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 Simulink Coder software generates 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 Simulink Coder 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

N	D	Ideal N/D	Zero	Floor	Undefined
-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.

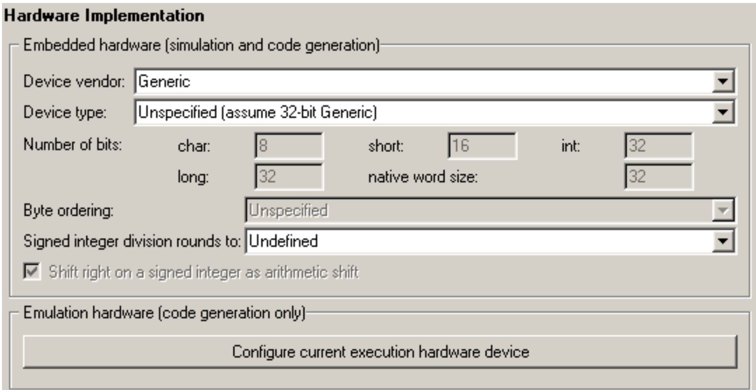
Select the option if the C compiler implements a signed integer right shift as an arithmetic right shift, and clear the option otherwise. An arithmetic right shift fills bits vacated by the right shift with the value of the most significant bit, which indicates the sign of the number in two's-complement notation. The option is selected by default. If your compiler handles right shifts as arithmetic shifts, this is the preferred setting.

- When the option is selected, the Simulink Coder software generates simple efficient code whenever the Simulink model performs arithmetic shifts on signed integers.
- When the option is cleared, the Simulink Coder software generates 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, by default the **Hardware Implementation** pane initially looks like this:



Click **Configure current execution hardware device**. The **Configure current execution hardware device** button disappears. The subpane then appears as shown in the first figure in this section. Save your model at this point to avoid redoing **Configure current execution hardware device** next time you access the **Hardware Implementation** pane.

Configure Production Hardware Characteristics

"Configure Test and Production Target Hardware" on page 6-3 documents the parameters available on the **Hardware Implementation** subpanes. This section describes considerations that apply only to the **Production hardware** subpane. When you use this subpane, 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 either the deployment target or the MATLAB host. The Simulink Coder code generator takes these differences into account when generating code.
- The Simulink product uses some of the information in the **Production hardware** subpane so that simulation without code generation gives the same results as executing generated code, including detecting error conditions that could arise on the target hardware, such as hardware overflow.
- The Simulink Coder software generates code that produces 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, you must 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 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, models that they reference must be configured to use identical hardware settings.

Configure Test Hardware Characteristics

"Configure Test and Production Target Hardware" on page 6-3 documents the parameters available on the **Hardware Implementation** subpanes. This section describes considerations that apply only to the **Test hardware** subpane.

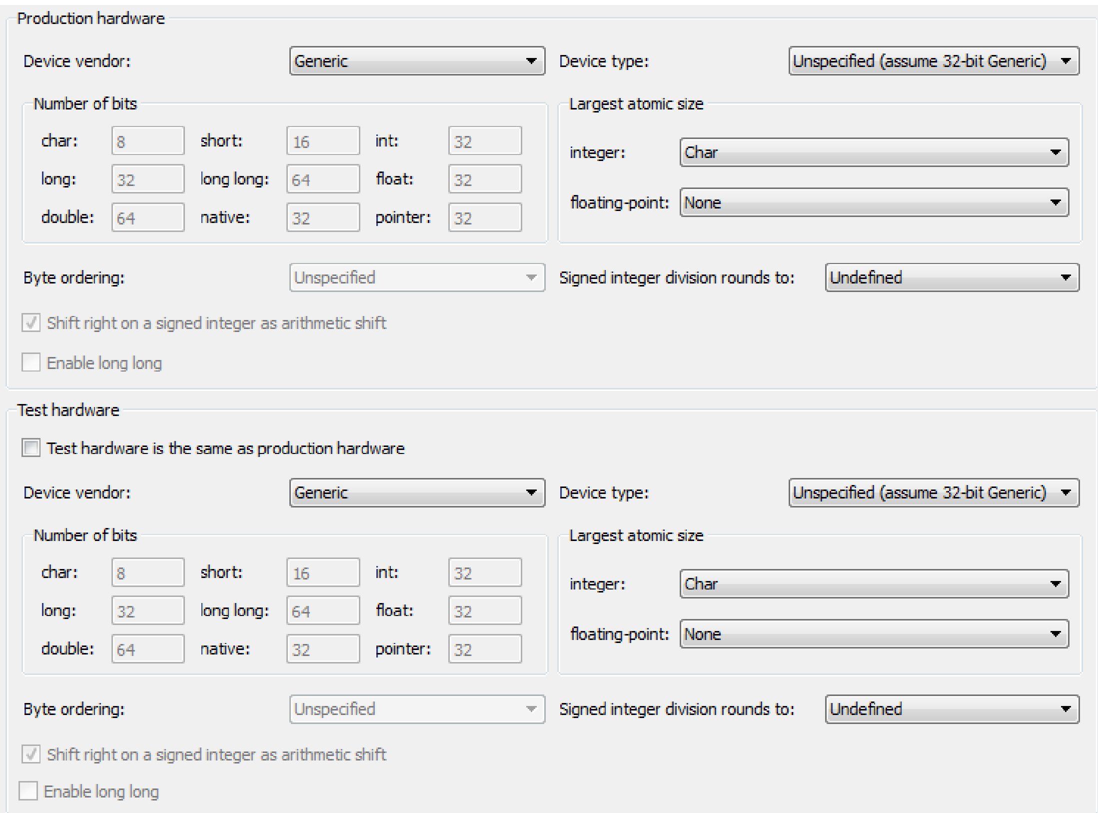
Note: If the **Test hardware** subpane contains a button labeled **Configure current execution hardware device**, see "Update Release 14 Hardware Configuration" on page 6-12, then continue from this point.

The default assumption is that the production target and test target are the same, so the **Test hardware** subpane by default does not need to specify anything and contains only a selected check box labeled **Test hardware is the same as production hardware**. Code generated under this configuration will be suitable for production use, or for testing in an environment identical to the production environment.

To generate code that runs on a test target for test purposes, but behaves as if it were running on a production target in a production application, you must specify the properties of both targets in the **Hardware Implementation** pane. The **Production hardware** subpane specifies production target hardware properties, as described previously. To specify test target properties:

1. Clear the **Test hardware is the same as production hardware** option in the **Test hardware** subpane.

   By default, the **Hardware Implementation** pane now looks like this:



2 In the **Test hardware** subpane, specify the properties of the test target, using the instructions in “Configure Test and Production Target Hardware” on page 6-3

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 both subpanes of the **Hardware Implementation** pane.

Parameters with only one possible value cannot be changed. Parameters that have more than one possible value provide a list of legal values. If you modify any 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.

If you do not display the **Test hardware** subpane, the Simulink and Simulink Coder software defaults every **Test hardware** parameter to have the same value as the corresponding **Production hardware** parameter. If you hide the **Test hardware** subpane after setting its values, the values that you specified will be lost. The underlying configuration parameters immediately revert to the values that they had when you exposed the subpane, and these values, rather than the values that you specified, will appear if you re-expose the subpane.

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 as well as code generation targets, artifacts used for simulation and Simulink Coder 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 Simulink Coder 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 6-19.

For more information about setting up a code generation folder, see “Control the Location for Code Generation Files” on page 6-21.

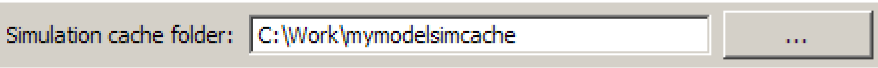
For more information about directly manipulating the MATLAB session parameters `CacheFolder` and `CodeGenFolder`, see "Override Build Folder Settings for Current Session" on page 6-23.

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,

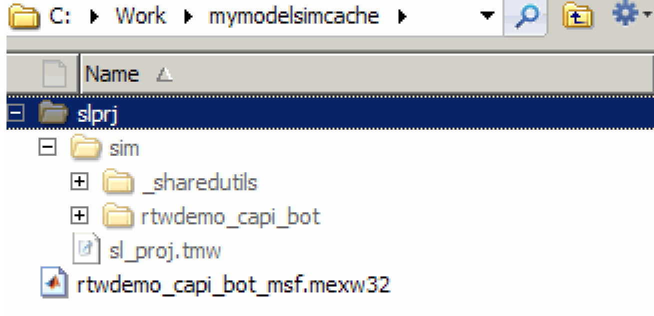
- You need to keep generated files separate from the models and other source materials used to generate them.
- You want to reuse or share previously-built simulation targets without having to set the current working folder back to a previous working folder.

The Simulink preference "**Simulation cache folder**" provides control over the output location for files generated by Simulink diagram updates. The preference appears in the Simulink Preferences Window, Main Pane, in the **File generation control** group. 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, for example:



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 follows:



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
```

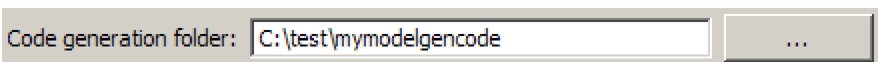
Also, you can choose to override the “**Simulation cache folder**” preference value for the current MATLAB session.

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 Simulink Coder 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,

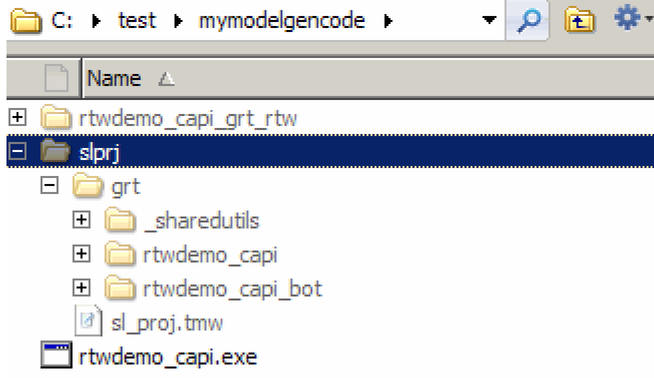
- You need to keep generated files separate from the models and other source materials used to generate them.
- You want to separate generated production code from generated simulation artifacts.

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 Simulink Preferences Window, Main Pane, in the **File generation control** group. 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, for example:



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 follows:



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 6-23.

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

- “Protect a Referenced Model” on page 7-2
- “Harness Model” on page 7-4
- “Protected Model Report” on page 7-5
- “Code Generation Support in a Protected Model” on page 7-6
- “Protected Model File” on page 7-7
- “Create a Protected Model” on page 7-9
- “Protected Model Creation Settings” on page 7-14
- “Test the Protected Model” on page 7-16
- “Save Base Workspace Definitions” on page 7-18
- “Package a Protected Model” on page 7-19
- “Specify Custom Obfuscator for Protected Model” on page 7-20

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 7-4.
- 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. Including code generation support with a protected model also allows the third party to simulate a model that references the protected model in Accelerator mode. For more information, see “Code Generation Support in a Protected Model” on page 7-6.
- If the Model block uses variants, only the active variant is protected. For more information, see “Set Up Model Variants”.
- 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 7-9
- 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.

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 7-9. You can use a harness model to test your protected model. For more information, see “Test the Protected Model” on page 7-16. 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

A protected model report can be generated when the protected model is created. The report is included as part of the protected model. It provides the receiver with the information to determine if the model can be run in the environment. 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 can support.
  - **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.

The protected model report is generated automatically when the protected model is created from the Simulink Editor. To generate a report when using the `Simulink.ModelReference.protect` function, set the '`Report`' option to `true`.

To view the protected model report, right-click the protected-model badge icon and select **Display Report**.

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 7-7 and "Protected Model Report" on page 7-5.

In the Create Protected Model dialog box, select **Generate code** check box. The appearance of the generated code is determined by the **Generated code 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 7-9.

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.

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` mode, after simulation, the following files are unpacked:

- `slprj/sim/`*`model`*`/*.h`
- `slprj/sim/`*`model`*`/`*`model`*`lib.a` (or *`model`*`lib.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`*`/`*`model`*`lib.a` (or *`model`*`lib.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`*`/_sharedutils/*`
- `slprj/`*`target`*`/`*`model`*`/tmwinternal/*`

For the protected model report, after building your model these files are unpacked (in addition to the preceding files):

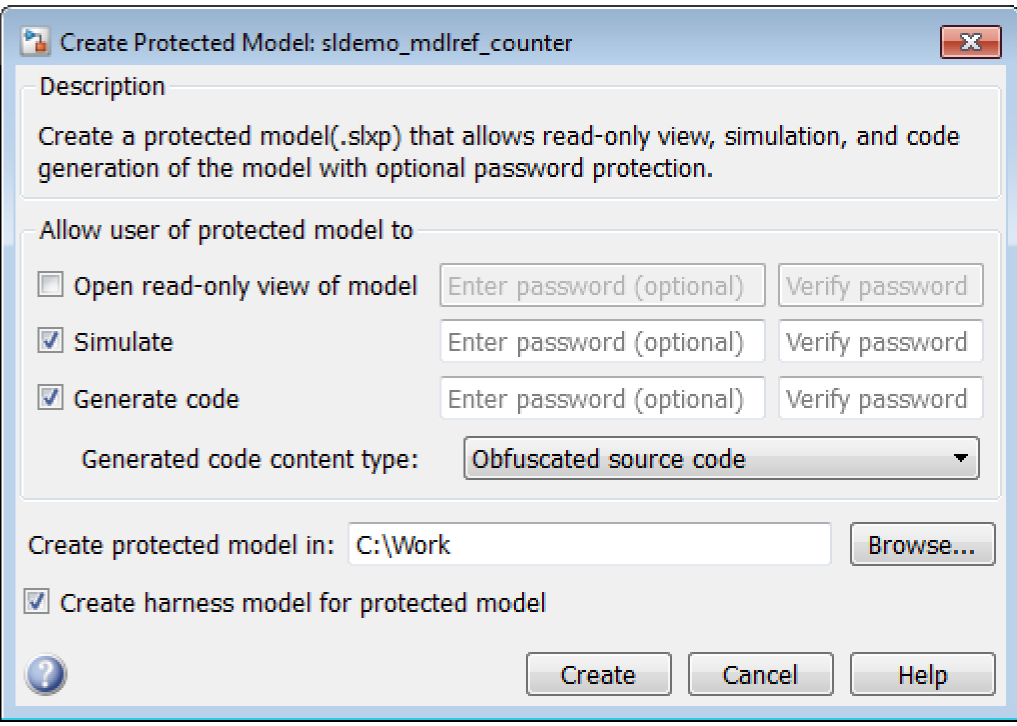
- `slprj/`*`target`*`/`*`model`*`/html/*`
- `slprj/`*`target`*`/`*`model`*`/buildinfo.mat`
- `slprj/`*`target`*`/_sharedutils/html/*`

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_mdlref_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**.



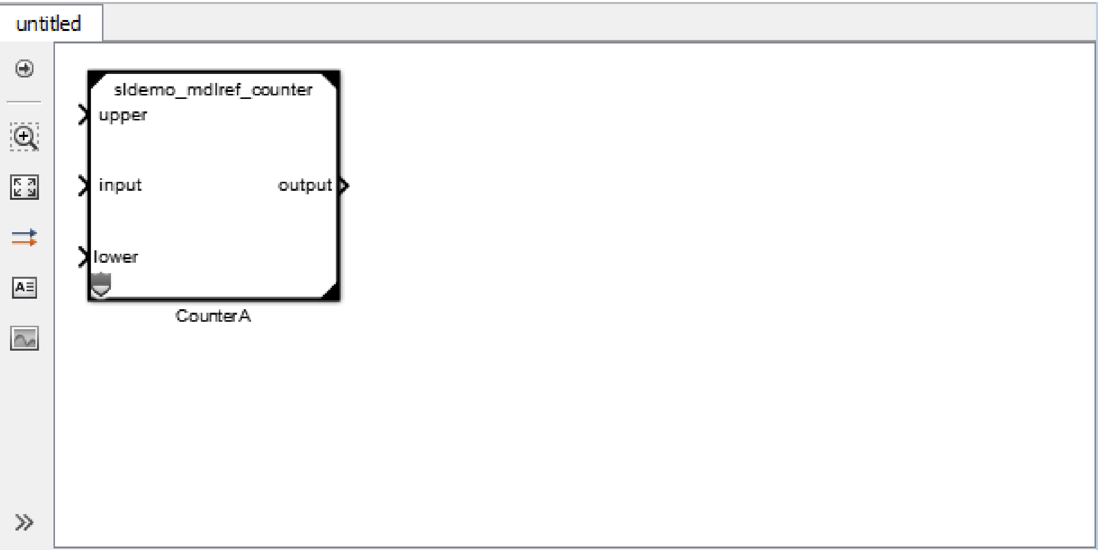
7 In the Create Protected Model dialog box, select the **Simulate** and **Generate code** check boxes. If you want to password-protect the functionality of the protected model, enter a password with a minimum of four characters. Each functionality can have a unique password.

8 From the **Generate code content type** list, select `Obfuscated source code` to conceal the source code purpose and logic of the protected model.

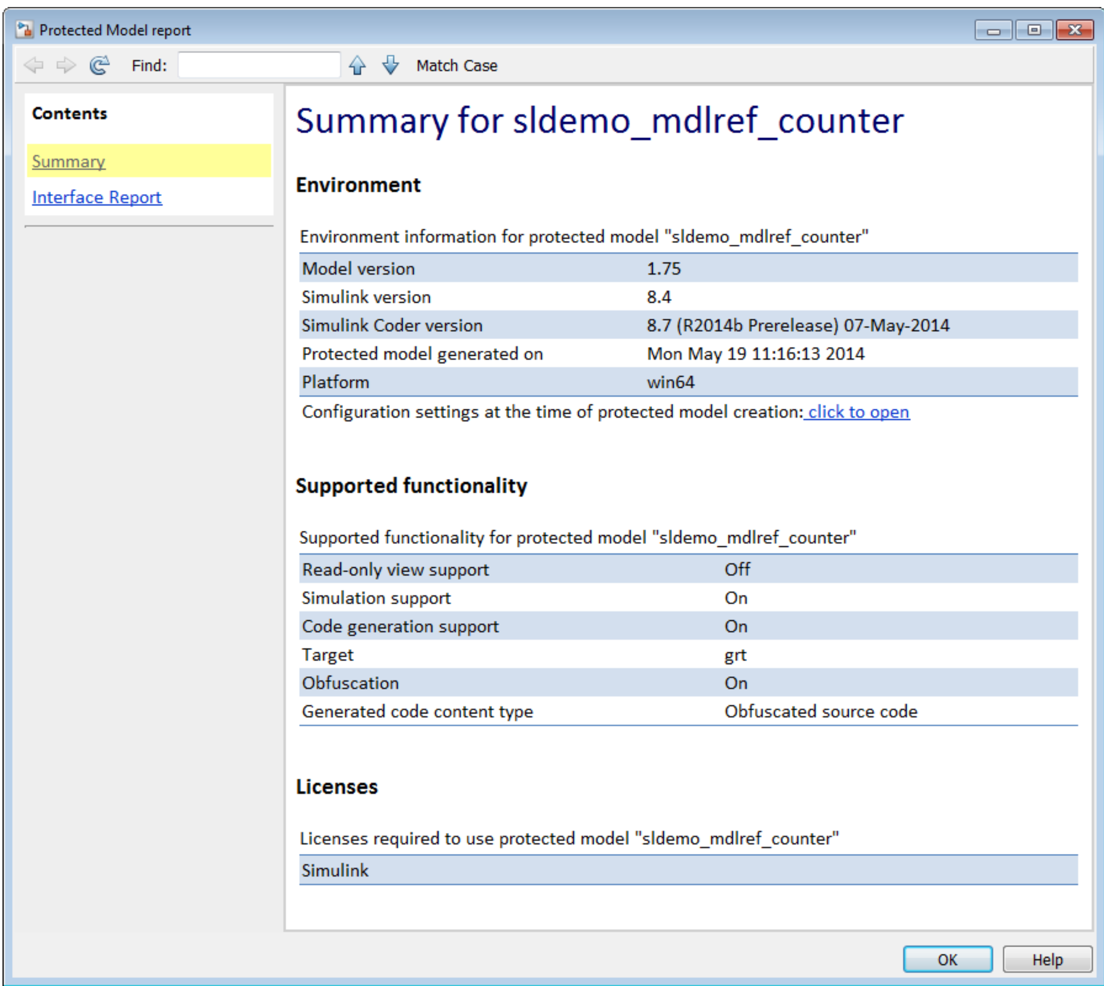
9 In the **Create protected model in** field, specify the folder path for the protected model. The default value is the current working folder.

10 To create a harness model for the protected model, select the **Create harness model for protected model** check box.

11 Click **Create**. An untitled harness model opens. It contains a model block, which refers to the protected model `sldemo_mdlref_counter.slxp`. The **Simulation mode** for the Model block is set to `Accelerator`. You cannot change the mode.



12 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 7-16
- “Package a Protected Model” on page 7-19

More About

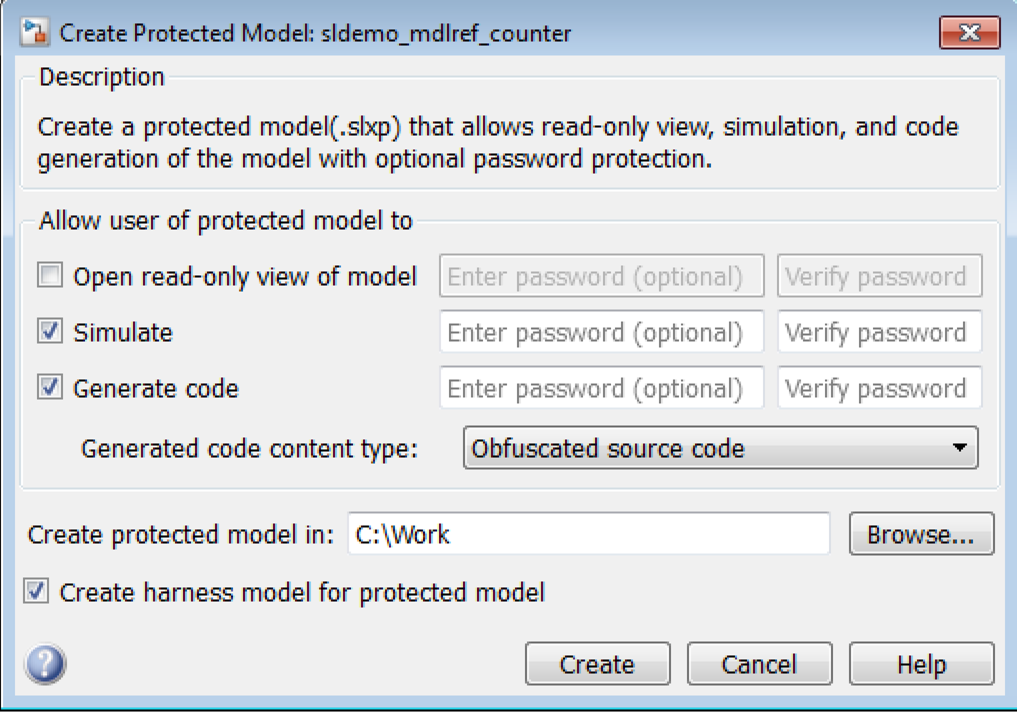
- “Code Generation Support in a Protected Model” on page 7-6
- “Protected Model Creation Settings” on page 7-14

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 allows 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.

Generate Code

The **Generate 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.
- Displays code in the build folder in obfuscated form.
- Determines the appearance of the generated code by the **Generated code content type** list. The options are:
  - `Binaries`
  - `Obfuscated source code` (default)
  - `Readable source code`, which also includes readable code comments

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_mdlref_basic` and the protected model, `sldemo_mdlref_counter.slxp`, which is created in “Create a Protected Model” on page 7-9.

1. If it is not already open, open `sldemo_mdlref_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 and set **Signal logging format** to `Dataset`. 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_mdlref_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_mdlref_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 "Workspace Variables in Model Explorer".

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 7-18.
- 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.

Data, Function, and File Definition

8

Data Representation

- "Enumerations" on page 8-2
- "Structure Parameters and Generated Code" on page 8-9
- "Parameters" on page 8-11
- "Signals" on page 8-46
- "States" on page 8-74
- "Data Stores" on page 8-85

Enumerations

In this section...

"About Enumerated Data Types" on page 8-2
"Default Code for an Enumerated Data Type" on page 8-2
"Specify Enumerated Data Type" on page 8-3
"Type Casting for Enumerations" on page 8-4
"Override Default Methods (Optional)" on page 8-5
"Enumerated Type Limitations" on page 8-8

About 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(Enumeration) BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Blue(2)
  end
end
```

For 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".

Default Code for an Enumerated Data Type

By default, enumerated data types in generated code are defined in the generated header file `model_types.h` for the model. For example, the default code for `BasicColors`, which is defined in the previous section, appears as follows:

```
#ifndef _DEFINED_TYPEDEF_FOR_BasicColors_
```

```
#define _DEFINED_TYPEDEF_FOR_BasicColors_

typedef enum {
  Red = 0,             /* Default value */
  Yellow = 1,
  Blue = 2,
} BasicColors;

#endif
```

Specify Enumerated Data Type

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.

For example, to specify a data type size of `int8`, specify this class definition:

```
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)
```

Type Casting for Enumerations

How Safe Casting Works

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.

Override Default Methods (Optional)

Every enumerated class has four associated static methods, which it inherits from *`Simulink.IntEnumType`*. You can optionally override these static methods to customize the behavior of an enumerated type. The methods are:

- `getDefaultValue` — Returns the default value of the enumerated data type.
- `getDescription` — Returns a description of the enumerated data type.
- `getHeaderFile` — Specifies a file where the type is defined for generated code.
- `addClassNameToEnumNames` — Specifies whether the class name becomes a prefix in code.

The first of these methods, `getDefaultValue`, is relevant to both simulation and code generation, and is described in "Specifying a Default Enumerated Value" in the Simulink documentation. The other three methods are relevant only to code generation, and are described in this section. To override the methods, include a customized version of the method in the enumerated class definition's `methods` section. If you do not want to override the default methods, omit the `methods` section entirely. The following table summarizes the four methods and the data to supply for each one:

Method	Purpose	Default Return	Custom Return
`getDefaultValue`	Returns the default value for the class, which must be an instance of the class.	The lexically first value in the enumeration.	An enumerated value in the class. See "Instantiate Enumerations".

Method	Purpose	Default Return	Custom Return
`getDescription`	Returns a string containing a description of the enumerated class.	`''`	A string that MATLAB accepts.
`getHeaderFile`	Returns a string containing the name of the header file	`''`	The name of the file that contains the enumerated type definition.
`addClassNameToEnumNames`	Returns a boolean value indicating whether to prefix the class name in generated code	`false`	`true` or `false`

Specifying a Description

To specify a description for an enumerated data type, include the following method in the enumerated class's `methods` section:

```
function retVal = getDescription()
% GETDESCRIPTION  Optional string to describe the data type.
  retVal = 'description';
end
```

Substitute a legal MATLAB string for *`description`*. The generated code that defines the enumerated type will include the specified description.

Specify a Header File

To prevent the declaration of an enumerated type from being embedded in the generated code, allowing you to provide the declaration in an external file, include the following method in the enumerated class's `methods` section:

```
function retVal = getHeaderFile()
% GETHEADERFILE  File where type is defined for generated code.
%   If specified, this file is #included  in the code.
%   Otherwise, the type is written out in the generated code.
retVal = 'filename';
end
```

Substitute a legal filename for *filename*. Be sure to provide a filename suffix, typically .h. Providing the method replaces the declaration that would otherwise have appeared in model_types.h with a #include statement like:

```
#include "imported_enum_type.h"
```

The getHeaderFile method does not create the declaration file itself. You must provide a file of the specified name that declares the enumerated data type.

Add Prefixes To Class Names

By default, enumerated values in generated code have the same names that they have in the enumerated class definition. Alternatively, the code can prefix every enumerated value in an enumerated class with the name of the class. This technique can be useful for preventing identifier conflicts or improving the clarity of the code. To specify class name prefixing, include the following method in an enumerated class's methods section:

```
function retVal = addClassNameToEnumNames()
% ADDCLASSNAMETOENUMNAMES  Control whether class name is added as
%   a prefix to enumerated names in the generated code.
%   By default the code does not use the class name as a prefix.
retVal = boolean;
end
```

Replace *boolean* with true to enable class name prefixing, or false to suppress prefixing without having to delete the method itself. If *boolean* is true, each enumerated value in the class appears in generated code as *EnumTypeName_EnumName*. For BasicColors, which was defined in “About Enumerated Data Types” on page 8-2, the data type definition with class name prefixing looks 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
```

In this example, the enumerated class name BasicColors appears as a prefix for each of the enumerated names. The definition is otherwise the same as it would be without name prefixing.

Enumerated Type Limitations

- Generated code does not support logging enumerated data.

Structure Parameters and Generated Code

In this section...
"About Structure Parameters and Generated Code" on page 8-9
"Configure Structure Parameters for Generated Code" on page 8-9
"Control Name of Structure Parameter Type" on page 8-10

About Structure Parameters and Generated Code

Structure parameters provide a way to improve generated code to use structures rather than multiple separate variables. You also have the option of configuring the appearance of a structure parameter in generated code.

For more information about structure parameters, see "Structure Parameters" in the Simulink documentation. For an example of how to convert a model that uses unstructured workspace variables to a model that uses structure parameters, see sldemo_applyVarStruct.

Configure Structure Parameters for Generated Code

By default, structure parameters do not appear in generated code. Structure parameters include numeric variables and the code generator inlines numeric values.

To make structure type definition appear in generated code for a structure parameter,

1. Create a `Simulink.Parameter` object.
2. Define the object value to be the parameter structure.
3. Define the object storage class to be a value other than `Auto`.

The code generator places a structure type definition or the tunable parameter structure in *`model`*`_types.h`. By default, the code generator identifies the type with a nondescriptive, automatically generated name, such as `struct_z98c0D2qc4btL`.

For information on how to control the naming of the type, see "Control Name of Structure Parameter Type" on page 8-10. For an example, see sldemo_applyVarStruct

Control Name of Structure Parameter Type

To control the naming of a structure parameter type, use a `Simulink.Bus` object to specify the data type of the `Simulink.Parameter` object.

1. Use Simulink.Bus.createObject to create a bus object with the same shape as the parameter structure. For example:

   ```
   busInfo=Simulink.Bus.createObject(ControlParam.Value);
   ```

2. Assign the bus object name to the data type property of the parameter object.

   ```
   ParamType=eval(busInfo.busName);
   ControlParam.DataType='Bus: ParamType';
   ```

Only `Simulink.Parameter` can accept the bus object name as a data type.

For an example, see sldemo_applyVarStruct

Parameters

In this section...
“About Parameters” on page 8-11
“Nontunable Parameter Storage” on page 8-12
“Tunable Parameter Storage” on page 8-14
“Tunable Parameter Storage Classes” on page 8-15
“Declare Tunable Parameters” on page 8-17
“Tunable Expressions” on page 8-21
“Linear Block Parameter Tunability” on page 8-25
“Configuration Parameter Quick Reference Diagram” on page 8-26
“Generated Code for Parameter Data Types” on page 8-27
“Tunable Workspace Parameter Data Type Considerations” on page 8-32
“Tune Parameters” on page 8-34
“Parameter Objects” on page 8-35

About Parameters

This section discusses how the Simulink Coder product generates parameter storage declarations, and how you can generate the storage declarations you need to interface block parameters to your code. For information about defining block parameters in Simulink models, see “Set Block Parameters”.

If you are using S-functions in your model and intend to tune their run-time parameters in the generated code, see “Tuning Run-Time Parameters” in the Simulink documentation. Note that

- Parameters must be numeric, logical, or character arrays.
- Parameters may not be sparse.
- Parameter arrays must not be greater than 2 dimensions.

For guidance on implementing a parameter tuning interface using a C API, see “ASAP2 Data Measurement and Calibration”.

Simulink external mode offers a way to monitor signals and modify parameter values while generated model code executes. However, external mode might not be the optimal

solution for your application. For example, the S-function target does not support external mode. For other targets, you might want your existing code to access parameters and signals of a model directly, rather than using the external mode communications mechanism. For information on external mode, see “Host/Target Communication”.

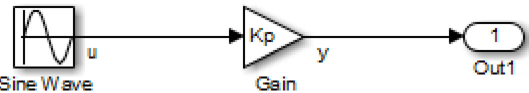
Nontunable Parameter Storage

When **Inline Parameters** is off (the default), the Simulink Coder product has control of parameter storage declarations and the symbolic naming of parameters in the generated code.

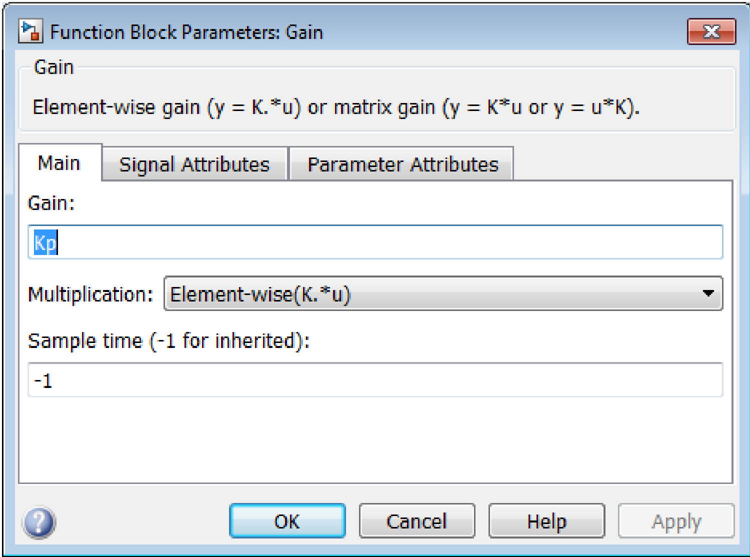
Nontunable parameters are stored as fields within `model_P` (formerly `rtP`), a model-specific global parameter data structure. The Simulink Coder product initializes each field of `model_P` to the value of the corresponding block parameter at code generation time.

When the **Inline parameters** option is on, block parameters are evaluated at code generation time, and their values appear as constants in the generated code, if possible (in certain circumstances, parameters cannot be inlined, and are then included in a constant parameter or model parameter structure.)

As an example of nontunable parameter storage, consider the following model.



The workspace variable `Kp` sets the gain of the `Gain` block.



Assume that Kp is nontunable and has a value of 5.0. The next table shows the variable declarations and the code generated for Kp in the noninlined and inlined cases.

The generated code does not preserve the symbolic name Kp. The noninlined code represents the gain of the Gain block as *model*_P.Gain_Gain. When Kp is noninlined, the parameter is tunable.

Inline Parameters	Generated Variable Declaration and Code

Off	`struct Parameters_non_tunable_sin {`
	`  real_T SineWave_Amp;`
	`  real_T SineWave_Bias;`
	`     real_T SineWave_Freq;`
	`     real_T SineWave_Phase;`
	`     real_T Kp;`
	`   };`
	`   .`
	`   .`
	`   .`
	`   Parameters_non_tunable_sin non_tunable_sin_P = {`
	`     1.0 , /* SineWave_Amp : '<Root>/Sine Wave' */`
	`     0.0 , /* SineWave_Bias : '<Root>/Sine Wave' */`
	`     1.0 , /* SineWave_Freq : '<Root>/Sine Wave' */`
	`     0.0 , /* SineWave_Phase : '<Root>/Sine Wave' */`

Inline Parameters	Generated Variable Declaration and Code
	5.0 /* Kp : '<Root>/Gain' */ }; . . . non_tunable_sin_Y.Out1 = rtb_u * non_tunable_sin_P.Kp;
On	non_tunable_sin_Y.Out1 = rtb_u * 5.0;

Tunable Parameter Storage

A *tunable* parameter is a block parameter whose value can be changed at run-time. A tunable parameter is inherently noninlined. Consequently, when **Inlined parameters** is off, parameters are members of *`model`*`_P`, and thus are tunable. A *tunable expression* is an expression that contains one or more tunable parameters.

When you declare a parameter tunable, you control whether or not the parameter is stored within *`model`*`_P`. You also control the symbolic name of the parameter in the generated code.

When you declare a parameter tunable, you specify

- The *storage class* of the parameter.

  The storage class property of a parameter specifies how the Simulink Coder product declares the parameter in generated code.

  The term “storage class,” as used in the Simulink Coder product, is not synonymous with the term *storage class specifier*, as used in the C language.
- A *storage type qualifier*, such as `const` or `volatile`. This is simply a string that is included in the variable declaration.
- (Implicitly) the symbolic name of the variable or field in which the parameter is stored. The Simulink Coder product derives variable and field names from the names of tunable parameters.

The Simulink Coder product generates a variable or `struct` storage declaration for each tunable parameter. Your choice of storage class controls whether the parameter is declared as a member of *`model`*`_P` or as a separate global variable.

You can use the generated storage declaration to make the variable visible to external legacy code. You can also make variables declared in your code visible to the generated code. You are responsible for linking your code to generated code modules.

You can use tunable parameters or expressions in your root model and in masked or unmasked subsystems, subject to certain restrictions. (See "Tunable Expressions" on page 8-21.)

Override Inlined Parameters for Tuning

When the **Inline parameters** option is selected, you can use the Model Parameter Configuration dialog box to remove individual parameters from inlining and declare them to be tunable. This allows you to improve overall efficiency by inlining most parameters, while at the same time retaining the flexibility of run-time tuning for selected parameters. Another way you can achieve the same result is by using Simulink data objects; see "Parameters" on page 8-11 for specific details.

The mechanics of declaring tunable parameters are discussed in "Declare Tunable Parameters" on page 8-17.

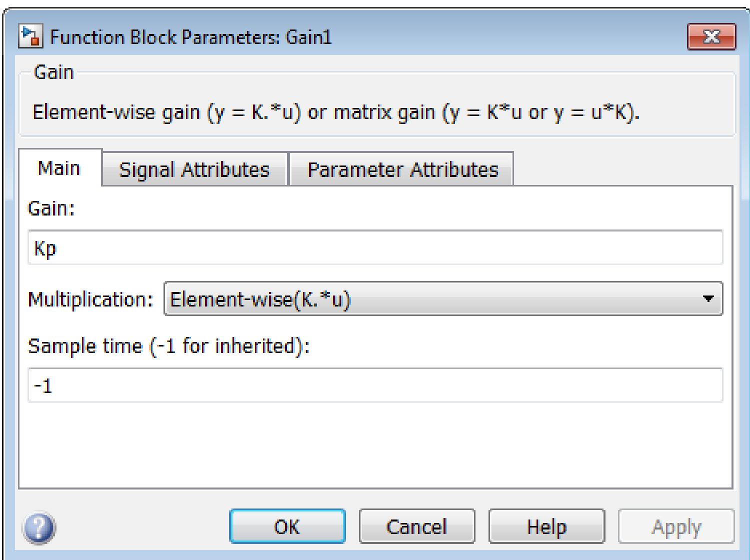
Tunable Parameter Storage Classes

The Simulink Coder product defines four storage classes for tunable parameters. You must declare a tunable parameter to have one of the following storage classes:

- `SimulinkGlobal (Auto)`: This is the default storage class. The Simulink Coder product stores the parameter as a member of *`model`*`_P`. Each member of *`model`*`_P` is initialized to the value of the corresponding workspace variable at code generation time.
- `ExportedGlobal`: The generated code instantiates and initializes the parameter and *`model`*`.h` exports it as a global variable. An exported global variable is independent of the *`model`*`_P` data structure. Each exported global variable is initialized to the value of the corresponding workspace variable at code generation time.
- `ImportedExtern`: *`model`*`_private.h` declares the parameter as an `extern` variable. Your code must supply the variable definition and initializer.
- `ImportedExternPointer`: *`model`*`_private.h` declares the variable as an `extern` pointer. Your code must supply the pointer variable definition and initializer.

The generated code for *`model`*`.h` includes *`model`*`_private.h` to make the `extern` declarations available to subsystem files.

As an example of how the storage class declaration affects the code generated for a parameter, consider the next figure.



The workspace variable `Kp` sets the gain of the `Gain` block. Assume that the value of `Kp` is 3.14. The following table shows the variable declarations and the code generated for the gain block when `Kp` is declared as a tunable parameter. An example is shown for each storage class.

Note The Simulink Coder product uses column-major ordering for two-dimensional signal and parameter data. When interfacing your hand-written code to such signals or parameters by using `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` declarations, make sure that your code observes this ordering convention.

The symbolic name `Kp` is preserved in the variable and field names in the generated code.

Storage Class	Generated Variable Declaration and Code
`SimulinkGlobal (Auto)`	`typedef struct _Parameters_tunable_sin Parameters_tunable_sin;`

Storage Class	Generated Variable Declaration and Code

	`struct _Parameters_tunable_sin {`
	`  real_T Kp;`
	`};`
	
	`Parameters_tunable_sin tunable_sin_P = {`
	`  3.14`
	`};`
	`.`
	`.`
	`tunable_sin_Y.Out1 = rtb_u *`
	`tunable_sin_P.Kp;`

`ExportedGlobal`	`real_T Kp = 3.14;`
	`.`
	`.`
	`tunable_sin_Y.Out1 = rtb_u * Kp;`

`ImportedExtern`	`extern real_T Kp;`
	`.`
	`.`
	`tunable_sin_Y.Out1 = rtb_u * Kp;`

`ImportedExtern`
`Pointer`
	`extern real_T *Kp;`
	`.`
	`.`
	`tunable_sin_Y.Out1 = rtb_u * (*Kp);`

Declare Tunable Parameters

- “Declare Workspace Variables as Tunable Parameters” on page 8-17
- “Declare New Tunable Parameters” on page 8-18
- “Declare Tunable Parameters Using Configuration Parameters” on page 8-18
- “Select Workspace Variables” on page 8-19
- “Create New Tunable Parameters” on page 8-20
- “Set Tunable Parameter Properties” on page 8-20
- “Remove Unused Tunable Parameters” on page 8-21

Declare Workspace Variables as Tunable Parameters

To declare tunable parameters,

1. Open the Model Parameter Configuration dialog box.
2. In the **Source list** pane, select one or more variables.
3. Click **Add to table** . The variables then appear as tunable parameters in the **Global (tunable) parameters** pane.
4. Select a parameter in the **Global (tunable) parameters** pane.
5. Select a storage class from the **Storage class** menu.
6. Optionally, select (or enter) a storage type qualifier, such as `const` or `volatile` for the parameter.
7. Click **Apply**, or click **OK** to apply changes and close the dialog box.

Declare New Tunable Parameters

To declare tunable parameters,

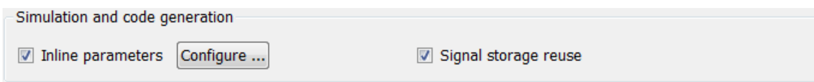
1. Open the Model Parameter Configuration dialog box.
2. In the **Global (tunable) parameters** pane, click **New**.
3. Specify a name for the parameter.
4. Select a storage class from the **Storage class** menu.
5. Optionally, select (or enter) a storage type qualifier, such as `const` or `volatile` for the parameter.
6. Click **Apply**, or click **OK** to apply changes and close the dialog box.

Declare Tunable Parameters Using Configuration Parameters

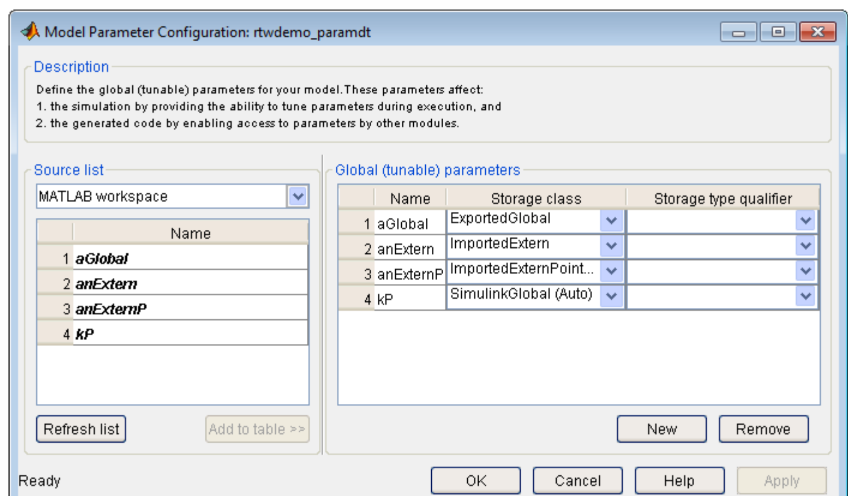
The Model Configuration Parameters dialog box lets you select base workspace variables and declare them to be tunable parameters in the current model. Using controls in the dialog box, you move variables from a source list to a global (tunable) parameter list for a model.

To open the dialog box,

1. Select the **Inline parameters** check box on the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box. This activates a **Configure** button, as shown below.



2 Click **Configure** to open the Model Configuration Parameters dialog box.



Note You cannot tune parameters within referenced models using the Model Configuration Parameters dialog box. See "Parameterize Model References" for tuning techniques that work with referenced models.

Select Workspace Variables

The **Source list** pane displays a menu and a scrolling table of numerical workspace variables. To select workspace variables,

1 From the menu, select the source of variables you want listed.

To List...	Choose...
Variables in the MATLAB workspace that have numeric values	`MATLAB workspace`
Only variables in the MATLAB workspace that have numeric values and are referenced by the model	`Referenced workspace variables`

A list of workspace variables appear in the **Source List** pane.

2. Select one or more variables from the source list. This enables the **Add to table** button.
3. Click **Add to table** to add the selected variables to the tunable parameters list in the **Global (tunable) parameters** pane. In the **Source list**, the names of variables added to the tunable parameters list are displayed in bold type (see the preceding figure).

> **Note:** If you selected a variable with a name that matches a block parameter that is not tunable and you click **Add to table** , a warning appears during simulation and code generation.

To update the list of variables to reflect the current state of the workspace, click **Refresh list** . For example, you might use **Refresh list** if you define or remove variables in the workspace while the Model Parameter Configuration dialog box is open.

Create New Tunable Parameters

To create a new tunable parameter,

1. In the **Global (tunable) parameters** pane, click **New**.
2. In the **Name** field, enter a name for the parameter.

   If you enter a name that matches the name of a workspace variable in the **Source list** pane, that variable is declared tunable and appears in italics in the **Source list**.
3. Click **Apply**.

The model does not need to be using a parameter before you create it. You can add references to the parameter later.

> **Note** If you edit the name of an existing variable in the list, you actually create a new tunable variable with the new name. The previous variable is removed from the list and loses its tunability (that is, it is inlined).

Set Tunable Parameter Properties

To set the properties of tunable parameters listed in the **Global (tunable) parameters** pane, select a parameter and then specify a storage class and, optionally, a storage type qualifier.

Property	Description
Storage class	Select one of the following to be used for code generation: • `SimulinkGlobal (Auto)` • `ExportedGlobal` • `ImportedExtern` • `ImportedExternPointer` See "Tunable Parameter Storage Classes" on page 8-15 for definitions.
Storage type qualifier	For variables with a storage class other than `SimulinkGlobal (Auto)`, you can add a qualifier (such as `const` or `volatile`) to the generated storage declaration. To do so, you can select a predefined qualifier from the list or add qualifiers not in the list. The code generator does not check the storage type qualifier for validity, and includes the qualifier string in the generated code without checking syntax .

Remove Unused Tunable Parameters

To remove unused tunable parameters from the table in the **Global (tunable) parameters** pane, click **Remove**. Removed variables are inlined if the **Inlined parameters** option is enabled.

Tunable Expressions

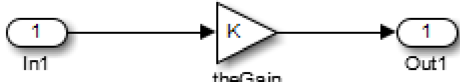
- "Tunable Expressions in Masked Subsystems" on page 8-21
- "Tunable Expression Limitations" on page 8-23

The Simulink Coder product supports the use of tunable variables in expressions. An expression that contains one or more tunable parameters is called a *tunable expression*.

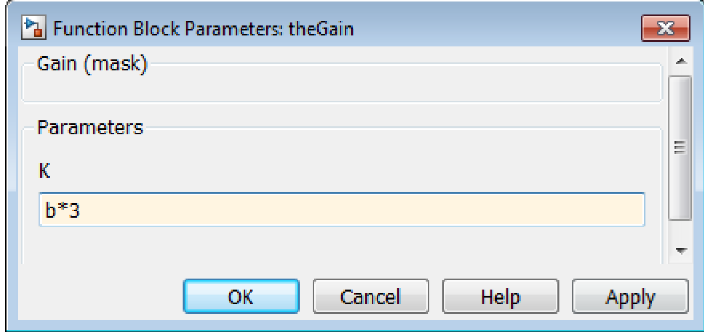
Tunable Expressions in Masked Subsystems

Tunable expressions are allowed in masked subsystems. You can use tunable parameter names or tunable expressions in a masked subsystem dialog box. When referenced in lower-level subsystems, such parameters remain tunable.

As an example, consider the masked subsystem in the next figure. The masked variable k sets the gain parameter of theGain.



Suppose that the base workspace variable b is declared tunable with SimulinkGlobal (Auto) storage class. The next figure shows the tunable expression b*3 in the subsystem's mask dialog box.



Tunable Expression in Subsystem Mask Dialog Box

The Simulink Coder product produces the following output computation for theGain. The variable b is represented as a member of the global parameters structure, *model*_P. (For clarity in showing the individual Gain block computation, expression folding is off in this example.)

```
/* Gain: '<S1>/theGain' */
  rtb_theGain_C = rtb_SineWave_n * ((subsys_mask_P.b * 3.0));

  /* Outport: '<Root>/Out1' */
  subsys_mask_Y.Out1 = rtb_theGain_C;
```

As this example shows, for GRT targets, the parameter structure is mangled to create the structure identifier *model*_P (subject to the identifier length constraint). This is done to avoid namespace clashes in combining code from multiple models using model reference. ERT-based targets provide ways to customize identifier names.

When expression folding is on, the above code condenses to

```
/* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<S1>/theGain'
   */
 subsys_mask_Y.Out1 = rtb_SineWave_n * ((subsys_mask_P.b * 3.0));
```

Expressions that include variables that were declared or modified in mask initialization code are *not* tunable.

As an example, consider the subsystem above, modified as follows:

- The mask initialization code is

  ```
  t = 3 * k;
  ```
- The parameter `k` of the `myGain` block is `4 + t`.
- Workspace variable `b = 2`. The expression `b * 3` is plugged into the mask dialog box as in the preceding figure.

Since the mask initialization code can run only once, `k` is evaluated at code generation time as

```
4 + (3 * (2 * 3) )
```

The Simulink Coder product inlines the result. Therefore, despite the fact that `b` was declared tunable, the code generator produces the following output computation for `theGain`. (For clarity in showing the individual Gain block computation, expression folding is off in this example.)

```
/* Gain Block: <S1>/theGain */
rtb_temp0 *= (22.0);
```

Tunable Expression Limitations

Currently, there are certain limitations on the use of tunable variables in expressions. When an unsupported expression is encountered during code generation a warning is issued and the equivalent numeric value is generated in the code. The limitations on tunable expressions are

- Complex expressions are not supported, except where the expression is simply the name of a complex variable.
- The use of certain operators or functions in expressions containing tunable operands is restricted. Restrictions are applied to four categories of operators or functions, classified in the following table:

<table>
<tr><th>Category</th><th>Operators or Functions</th></tr>
<tr><td>1</td><td><code>+ - .* ./ < > <= >= == ~= & |</code></td></tr>
<tr><td>2</td><td><code>* /</code></td></tr>
<tr><td>3</td><td><code>abs</code>, <code>acos</code>, <code>asin</code>, <code>atan</code>, <code>atan2</code>, <code>boolean</code>, <code>ceil</code>, <code>cos</code>, <code>cosh</code>, <code>exp</code>, <code>floor</code>, <code>log</code>, <code>log10</code>, <code>sign</code>, <code>sin</code>, <code>sinh</code>, <code>sqrt</code>, <code>tan</code>, <code>tanh</code>,</td></tr>
<tr><td>4</td><td><code>single</code>, <code>int8</code>, <code>int16</code>, <code>int32</code>, <code>uint8</code>, <code>uint16</code>, <code>uint32</code></td></tr>
<tr><td>5</td><td><code>: .^ ^ [] {} . \ .\ ' .' ; ,</code></td></tr>
</table>

The rules applying to each category are as follows:

- Category 1 is unrestricted. These operators can be used in tunable expressions with a combination of scalar or vector operands.
- Category 2 operators can be used in tunable expressions where at least one operand is a scalar. That is, scalar/scalar and scalar/matrix operand combinations are supported, but not matrix/matrix.
- Category 3 lists functions that support tunable arguments. Tunable arguments passed to these functions retain their tunability. Tunable arguments passed to other functions lose their tunability.
- Category 4 lists the casting functions that do not support tunable arguments. Tunable arguments passed to these functions lose their tunability.

  Note: The Simulink Coder product casts values using MATLAB typecasting rules. The MATLAB typecasting rules are different from C code typecasting rules. For example, using the MATLAB typecasting rules, `int8(3.7)` returns the result `4`, while in C code `int8(3.7)` returns the result `3`.

- Category 5 operators are not supported.

- Mask parameters are not tunable if they are hidden or disabled.
- Expressions that include variables that were declared or modified in mask initialization code are *not* tunable.
- The Fcn block does not support tunable expressions in code generation.
- Model workspace parameters can take on only the `Auto` storage class, and thus are not tunable. See “Parameterize Model References” for tuning techniques that work with referenced models.

- Non-double expressions are not supported.
- Blocks that access parameters only by address support the use of tunable parameters, if the parameter expression is a simple variable reference. When an operation such as a data type conversion or a math operation is applied, the Simulink Coder product creates a nontrivial expression that cannot be accessed by address, resulting in an error during the build process.

Linear Block Parameter Tunability

The following blocks have a `Realization` parameter that affects the tunability of their parameters:

- Transfer Fcn
- State-Space
- Discrete State-Space

The `Realization` parameter must be set by using the MATLAB `set_param` function, as in the following example.

```
set_param(gcb,'Realization','auto')
```

The following values are defined for the Realization parameter:

- `general`: The block's parameters are preserved in the generated code, permitting parameters to be tuned.
- `sparse`: The block's parameters are represented in the code by transformed values that increase the computational efficiency. Because of the transformation, the block's parameters are not tunable.
- `auto`: This setting is the default. A `general` realization is used if one or more of the block's parameters are tunable. Otherwise `sparse` is used.

Note To tune the parameter values of a block of one of the above types without restriction during an external mode simulation, you must set Realization to `general`.

Code Reuse for Subsystems with Mask Parameters

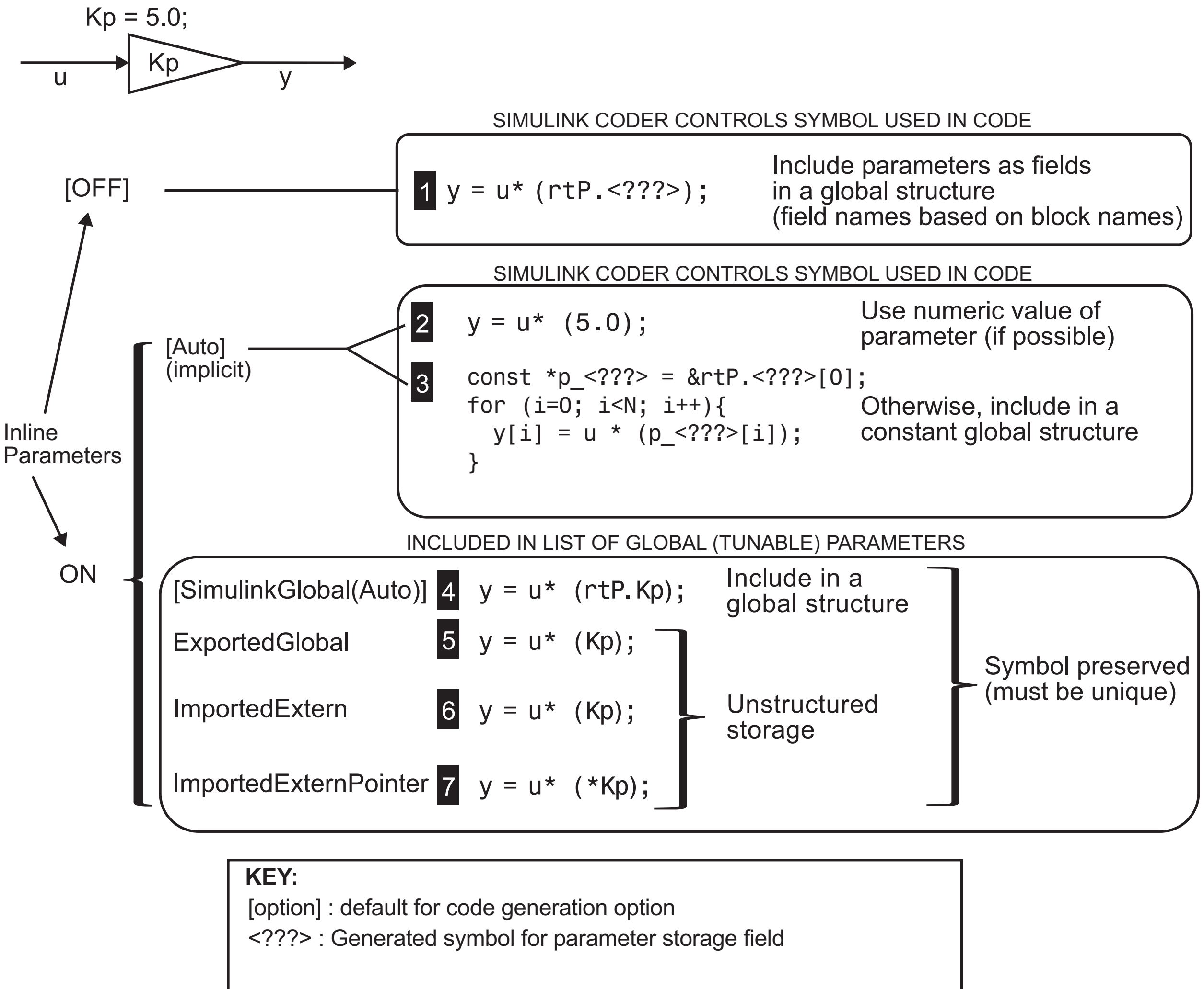
The Simulink Coder product can generate reusable (reentrant) code for a model containing identical atomic subsystems. Selecting the `Reusable function` option

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 Simulink Coder product will generate separate functions for A and B.

Configuration Parameter Quick Reference Diagram

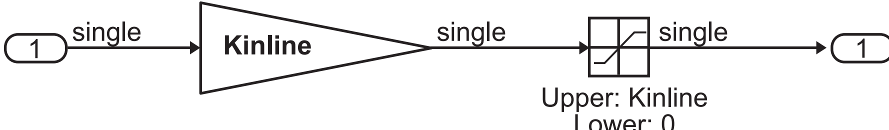
The next figure shows the code generation and storage class options that control the representation of parameters in generated code.

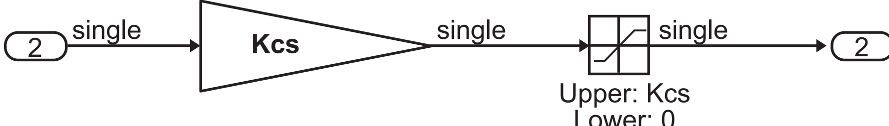


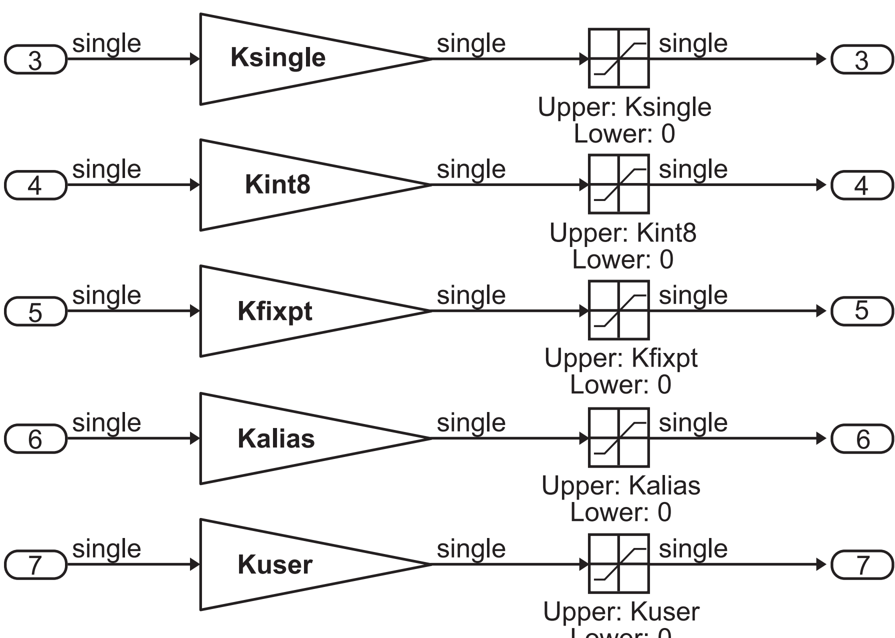
Generated Code for Parameter Data Types

For an example of the code generated from Simulink parameters with different data types, run the model rtwdemo_paramdt. This model shows options that are available for controlling the data type of tunable parameters in the generated code. The model's

subsystem includes several instances of Gain blocks feeding Saturation blocks. Each pair of blocks uses a workspace variable of a particular data type, as shown in the next figure.


Inlined parameters (InLineParameters ON + Auto storage class)
==> numeric value inlined
1
single
Kinline
single
single
1
Upper: Kinline
Lower: 0


Double-precision (context-sensitive) parameters
==> tunable parameter inherits data type from run-time parameter
2
single
Kcs
single
single
2
Upper: Kcs
Lower: 0


Tunable parameters with explicit data type specification
==> parameter is cast to run-time parameter data type in generated code
3
single
Ksingle
single
single
3
Upper: Ksingle
Lower: 0
4
single
Kint8
single
single
4
Upper: Kint8
Lower: 0
5
single
Kfixpt
single
single
5
Upper: Kfixpt
Lower: 0
6
single
Kalias
single
single
6
Upper: Kalias
Lower: 0
7
single
Kuser
single
single
7
Upper: Kuser
Lower: 0

The Simulink engine initializes the parameters in the model by executing the script `rtwdemo_paramdt_data.m`. You can view the initialization script and inspect the workspace variables in Model Explorer by double-clicking the yellow boxes in the model.

In the model, note that the **Inline parameters** option on the **Optimization** > **Signals and Parameters** pane of the **Configuration Parameters** dialog box is selected. The **Model Parameter Configuration** dialog box reveals that base workspace variables (with the exception of `Kinline`) have their **Storage class** property set to `ExportedGlobal`. Consequently, `Kinline` is a nontunable parameter while the remaining variables are tunable parameters.

To generate code for the model, double-click the blue boxes. The following table shows both the MATLAB code used to initialize parameters and the code generated for each parameter in the `rtwdemo_paramdt` model.

Parameter & MATLAB Code	Generated Variable Declaration and Code

Kinline
 	
`Kinline = 2;`
	`rtb_Gain1 = rtwdemo_paramdt_U.In1 * 2.0F;`
	`.`
	`.`
	`rtwdemo_paramdt_Y.Out1 = rt_SATURATE(rtb_Gain1, 0.0F, 2.0F);`

Kcs
 	
`Kcs = 3;`
	`real32_T Kcs = 3.0F;`
	`.`
	`.`
	`rtb_Gain1 = rtwdemo_paramdt_U.In2 * Kcs;`
	`.`
	`.`
	`rtwdemo_paramdt_Y.Out2 = rt_SATURATE(rtb_Gain1, 0.0F, Kcs);`

Ksingle
 	
`Ksingle = single(4);`
	`real32_T Ksingle = 4.0F;`
	`.`
	`.`
	`rtb_Gain1 = rtwdemo_paramdt_U.In3 * Ksingle;`
	`.`
	`.`
	`rtwdemo_paramdt_Y.Out3 = rt_SATURATE(rtb_Gain1, 0.0F, Ksingle);`

Kint8
 	
`Kint8 = int8(5);`
	`int8_T Kint8 = 5;`
	`.`
	`.`
	`rtb_Gain1 = rtwdemo_paramdt_U.In4 * ((real32_T)( Kint8 ));`
	`.`
	`.`
	`rtwdemo_paramdt_Y.Out4 = rt_SATURATE(rtb_Gain1, 0.0F,`
	`  ((real32_T)( Kint8 )));`

Kfixpt
 	
`Kfixpt = Simulink.Parameter;`
`Kfixpt.Value = 6;`
`Kfixpt.DataType = ...`
` 'fixdt(true, 16, 2^-5, 0)';`
`Kfixpt.CoderInfo.StorageClass = ...`
` 'ExportedGlobal';`
	`int16_T Kfixpt = 192;`
	`.`
	`.`
	`rtb_Gain1 = rtwdemo_paramdt_U.In5 *`
	`  (((real32_T)ldexp((real_T)Kfixpt, -5)));`
	`.`
	`.`
	`rtwdemo_paramdt_Y.Out5 = rt_SATURATE(rtb_Gain1, 0.0F,`
	`  (((real32_T)ldexp((real_T)Kfixpt, -5))));`

<table>
<tr><th>Parameter & MATLAB Code</th><th>Generated Variable Declaration and Code</th></tr>
<tr><td>Kalias

aliasType = ...
 Simulink.AliasType('single');
Kalias = Simulink.Parameter;
Kalias.Value = 7;
Kalias.DataType = 'aliasType';
Kalias.CoderInfo.StorageClass = ...
 'ExportedGlobal';</td><td>typedef real32_T aliasType;
.
.
aliasType Kalias = 7.0F;
.
.
rtb_Gain1 = rtwdemo_paramdt_U.In6 * Kalias;
.
.
rtwdemo_paramdt_Y.Out6 = rt_SATURATE(rtb_Gain1, 0.0F, Kalias);</td></tr>
<tr><td>Kuser

userType = Simulink.NumericType;
userType.DataTypeMode = ...
 'Fixed-point: slope and bias scaling';
userType.Slope = 2^-3;
userType.isAlias = true;
Kuser = Simulink.Parameter;
Kuser.Value = 8;
Kuser.DataType = 'userType';
Kuser.CoderInfo.StorageClass = ...
 'ExportedGlobal';</td><td>typedef int16_T userType;
.
.
userType Kuser = 64;
.
.
rtb_Gain1 = rtwdemo_paramdt_U.In7 *
  (((real32_T)ldexp((real_T)Kuser, -3)));
.
.
rtwdemo_paramdt_Y.Out7 = rt_SATURATE(rtb_Gain1, 0.0F,
  (((real32_T)ldexp((real_T)Kuser, -3))));</td></tr>
</table>

The salient features of the code generated for this model are as follows:

- The Simulink Coder product inlines nontunable parameters, for example, `Kinline`. However, the product does not inline tunable parameters, such as `Kcs`, `Ksingle`, and `Kint8`.
- The Simulink engine treats tunable parameters of data type `double` in a context-sensitive manner, such that the parameter inherits its data type from the context in which the block uses it. For example, `Kcs` inherits a `single` data type from the Gain block's input signal.
- If a parameter's data type matches that of the block's run-time parameter, the block can use the tunable parameter without transformation. Consequently, the Simulink Coder product need not cast the parameter from one data type to another, as illustrated by `Ksingle` and `Kalias`. However, if a parameter's data type does not match that of the block's run-time parameter, the block cannot readily compute its output. In this case, the product casts parameters to the relevant data type. For example, `Kint8`, `Kfixpt`, and `Kuser` require casts to a `single` data type for compatibility with the input signals to the Gain and Saturation blocks.
- If you are using an ERT target and a parameter specifies a data type alias, for example, created by an instance of the `Simulink.AliasType` class, its variable definition in the generated code uses the alias data type. For example, the Simulink Coder product declares `Kalias` and `Kuser` to be of data types `aliasType` and `userType`, respectively.

- If a parameter specifies a fixed-point data type, the Simulink Coder product initializes its value in the generated code to the value of Q computed from the expression V = SQ + B (see the Fixed-Point Designer documentation for more information about fixed-point semantics and notation), where
  - V is a real-world value
  - Q is an integer that encodes V
  - S is the slope
  - B is the bias

  For example, `Kfixpt` has a real-world value of 6, slope of 2^{-5}, and bias of 0. Consequently, the product declares the value of `Kfixpt` to be 192.

Tunable Workspace Parameter Data Type Considerations

If you are using tunable workspace parameters, you need to be aware of potential issues regarding data types. A workspace parameter is tunable when the following conditions exist:

- You select the **Inline parameters** option on the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box
- The parameter has a storage class other than `Auto`

When generating code for tunable workspace parameters, the Simulink Coder product checks and compares the data types used for a particular parameter in the workspace and in Block Parameter dialog boxes.

If...	The Simulink Coder Product...
The data types match	Uses that data type for the parameter in the generated code.

You do not explicitly specify a data type other than `double` in the workspace	Uses the data type specified by the block in the generated code. If multiple blocks share a parameter, they must specify the same data type. If the data type varies between blocks, the product generates an error similar to the following:
	`Variable 'K' is used in incompatible ways`
	`in the dialog fields of the following:`
	`cs_params/Gain, cs_params/Gain1. The`
	`variable'value is being used both directly`

If...	The Simulink Coder Product...
	`and after a transformation. Only one of these usages is permitted for any given variable.`
You explicitly specify a data type other than `double` in the workspace	Uses the data type from the workspace for the parameter. The block typecasts the parameter to the block specific data type before using it.

Guidelines for Specifying Data Types

The following table provides guidelines on specifying data types for tunable workspace parameters.

If You Want to...	Then Specify Data Types in...
Minimize memory usage (`int8` instead of `single`)	The workspace explicitly
Avoid typecasting	Blocks only
Interface to legacy or custom code	The workspace explicitly
Use the same parameter for multiple blocks that specify different data types	The workspace explicitly

The Simulink Coder product enforces limitations on the use of data types other than `double` in the workspace, as explained in "Limitations on Data Type Specifications in Workspace" on page 8-33.

Limitations on Data Type Specifications in Workspace

When you explicitly specify a data type other than `double` in the workspace, blocks typecast the parameter to a corresponding data type. This is an issue for blocks that use pointer access for their parameters. Blocks cannot use pointer access if they need to typecast the parameter before using it (because of a data type mismatch). Another case in which this occurs is for workspace variables with bias or fractional slope. Two possible solutions to these problems are

- Remove the explicit data type specification in the workspace for parameters used in such blocks.
- Modify the block so that it uses the parameter with the same data type as specified in the workspace. For example, the Lookup Table block uses the data types of its input signal to determine the data type that it uses to access the `X-breakpoint` parameter.

You can prevent the block from typecasting the run-time parameter by converting the input signal to the data type used for `X-breakpoints` in the workspace. (Similarly, the output signal is used to determine the data types used to access the lookup table `Y` data.)

Tune Parameters

- "Tune Parameters from the Command Line" on page 8-34
- "Interfaces for Tuning Parameters" on page 8-35

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 "Configure Parameter Objects for Code Generation" on page 8-36 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` (string) is a comma-separated list of variable names.
- `set_param(gcs, 'TunableVarsStorageClass', str)`

  The argument `str` (string) 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 (string) 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')
```

Interfaces for Tuning Parameters

The Simulink Coder product includes

- Support for developing a Target Language Compiler API for tuning parameters independent of external mode. See “Parameter Functions” in the Target Language Compiler documentation for information.
- A C application program interface (API) for tuning parameters independent of external mode. See “Data Interchange Using the C API” for information.
- An interface for exporting ASAP2 files, which you customize to use parameter objects. For details, see “ASAP2 Data Measurement and Calibration”.

Parameter Objects

- “About Parameter Objects for Code Generation” on page 8-36
- “Use Parameter Objects for Code Generation” on page 8-36
- “Configure Parameter Objects for Code Generation” on page 8-36
- “Storage Classes for Parameter Objects” on page 8-37
- “Configure Parameter Objects from Command Line” on page 8-37
- “Configure Parameter Objects Using Model Explorer” on page 8-39
- “Parameter Object Configuration Quick Reference Diagram” on page 8-42

- “Resolve Conflicts in Parameter Object Configurations” on page 8-43

About Parameter Objects for Code Generation

Within the class hierarchy of Simulink data objects, the Simulink product provides a class that is designed as base class for parameter storage. This topic explains how to use parameter objects in code generation.

The `CoderInfo` properties of parameter objects are used by the Simulink Coder product during code generation. These properties let you assign storage classes to the objects, thereby controlling how the generated code stores and represents parameters.

The Simulink Coder build process also writes information about the properties of parameter objects to the *`model`*`.rtw` file. This information, formatted as `Object` records, is accessible to Target Language Compiler programs. For general information on `Object` records, see “Data Object Information in model.rtw”.

Before using Simulink parameter objects with the Simulink Coder product, read the discussion of Simulink data objects in the Simulink documentation.

Use Parameter Objects for Code Generation

The general procedure for using parameter objects in code generation is as follows:

1. Define a subclass of `Simulink.Parameter`.
2. Instantiate parameter objects from your subclass and set their properties from the command line or by using Model Explorer.
3. Use the objects as parameters within your model.
4. Generate code and build your target executable.

Configure Parameter Objects for Code Generation

In configuring parameter objects for code generation, you use the following code generation and parameter object properties:

- The **Inline parameters** option (see “Parameters” on page 8-11).
- Parameter object properties:
    - `Value`. The numeric value of the object, used as an initial (or inlined) parameter value in generated code.

- `DataType`. The data type of the object. This can be a Simulink numeric data type, including a fixed-point, user-defined, or alias data type.
- `CoderInfo.StorageClass`. Controls the generated storage declaration and code for the parameter object.

Other parameter object properties (such as user-defined properties of classes derived from `Simulink.Parameter`) do not affect code generation.

Storage Classes for Parameter Objects

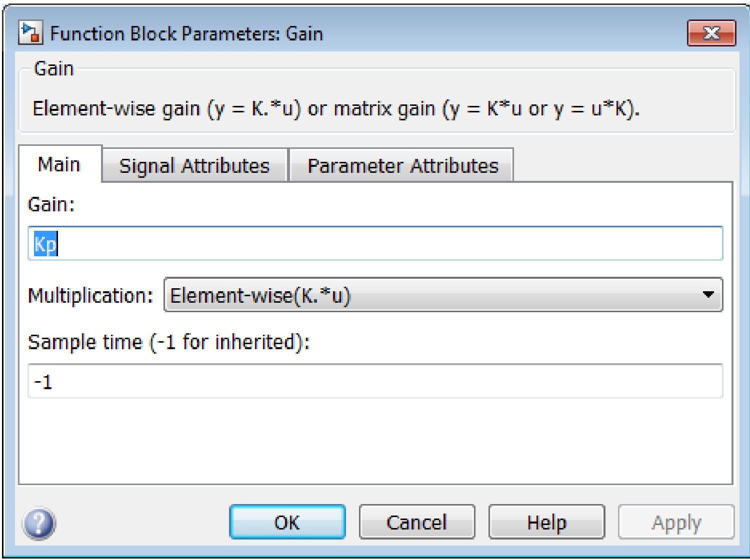
The Simulink Coder product generates code and storage declarations based on the `CoderInfo.StorageClass` property of the parameter object. The logic is as follows:

- If the storage class is `'Auto'` (the default), the parameter object is inlined (if possible), using the `Value` property.
- For storage classes other than `'Auto'`, the parameter object is handled as a tunable parameter.
  - A global storage declaration is generated. You can use the generated storage declaration to make the variable visible to your hand-written code. You can also make variables declared in your hand-written code visible to the generated code.
  - The symbolic name of the parameter object is generally preserved in the generated code.

See the table in “Configure Parameter Objects Using Model Explorer” on page 8-39 for examples of code generated for possible settings of `CoderInfo.StorageClass`.

Configure Parameter Objects from Command Line

In this section, the Gain block computations of the model shown in the next figure are used as an example of how the Simulink Coder build process generates code for a parameter object.



Model Using Parameter Object Kp As Block Parameter

In this model, `Kp` sets the gain of the Gain block.

To configure a parameter object such as `Kp` for code generation:

1 Instantiate a `Simulink.Parameter` object called `Kp`. In this example, the parameter object is an instance of the example class `SimulinkDemos.Parameter`, which is provided with the Simulink product.

```
Kp = Simulink.Parameter
Kp =
Simulink.Parameter
          Value: 5
        CoderInfo: [1x1 Simulink.ParamCoderInfo]
    Description: ''
       DataType: 'auto'
            Min: []
            Max: []
       DocUnits: ''
     Complexity: 'real'
     Dimensions: '[1x1]'
```

Make sure that the name of the parameter object matches the desired block parameter in your model. This enables the Simulink engine to associate the parameter name with the corresponding object. In the preceding model, the Gain block parameter `Kp` resolves to the parameter object `Kp`.

2 Set the object properties you need. You can do this by using the Model Explorer, or you can assign properties by using MATLAB commands, as follows:

   - To specify the `Value` property, type

     ```
     Kp.Value = 5.0;
     ```

   - To specify the storage class of for the parameter, set the `CoderInfo.StorageClass` property, for example:

     ```
     Kp.CoderInfo.StorageClass = 'ExportedGlobal';
     ```

   The `CoderInfo` parameters are now

   ```
   Kp.CoderInfo
   Simulink.ParamCoderInfo
             StorageClass: 'ExportedGlobal'
                    Alias: ''
       CustomStorageClass: 'Default'
         CustomAttributes: [1x1
   SimulinkCSC.AttribClass_Simulink_Default]
   ```

Configure Parameter Objects Using Model Explorer

If you prefer, you can create and modify attributes of parameter objects using the Model Explorer. This lets you see the attributes of a parameter in a dialog box, and alleviates the need to remember and type field names. Do the following to instantiate `Kp` and set its attributes from Model Explorer:

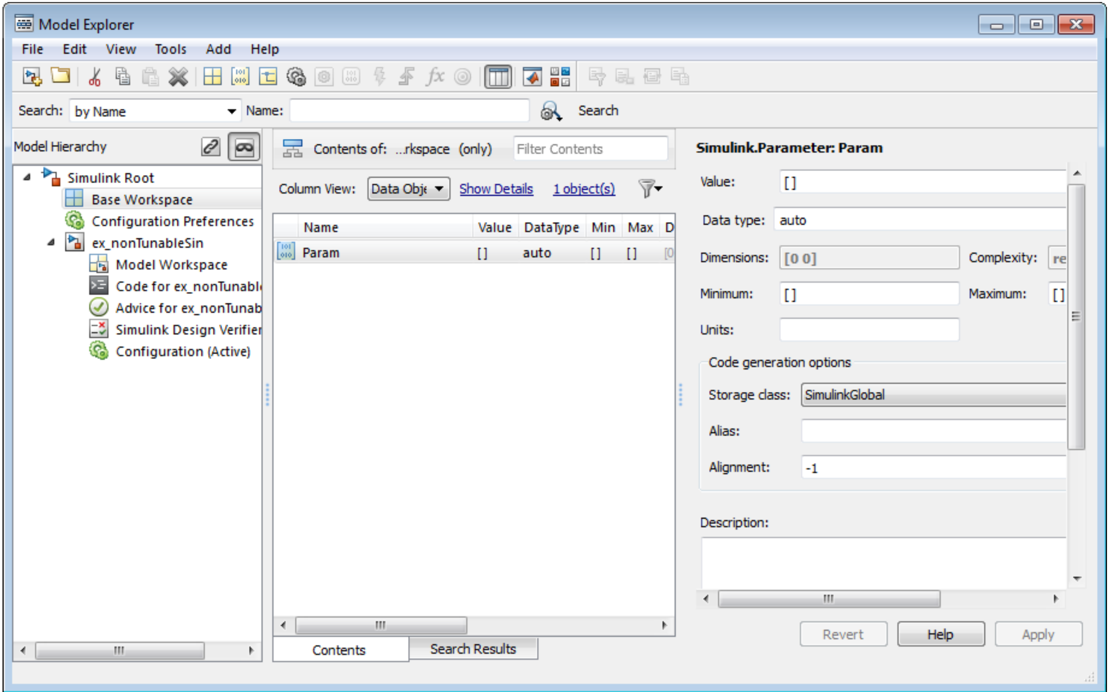
1 Choose **Model Explorer** from the **View** menu.

   Model Explorer opens or activates if it already was open.

2 Select `Base Workspace` in the **Model Hierarchy** pane.

3 Select **Simulink Parameter** from the **Add** menu.

   A new parameter named `Param` appears in the **Contents** pane.



4 To set `Kp.Name` in the Model Explorer:

  a Click the word `Param` in the **Name** column to select it.

  b Rename it by typing `Kp` in place of `Param`.

  c Press **Enter** or **Return**.

5 To set `Kp.Value` in Model Explorer:

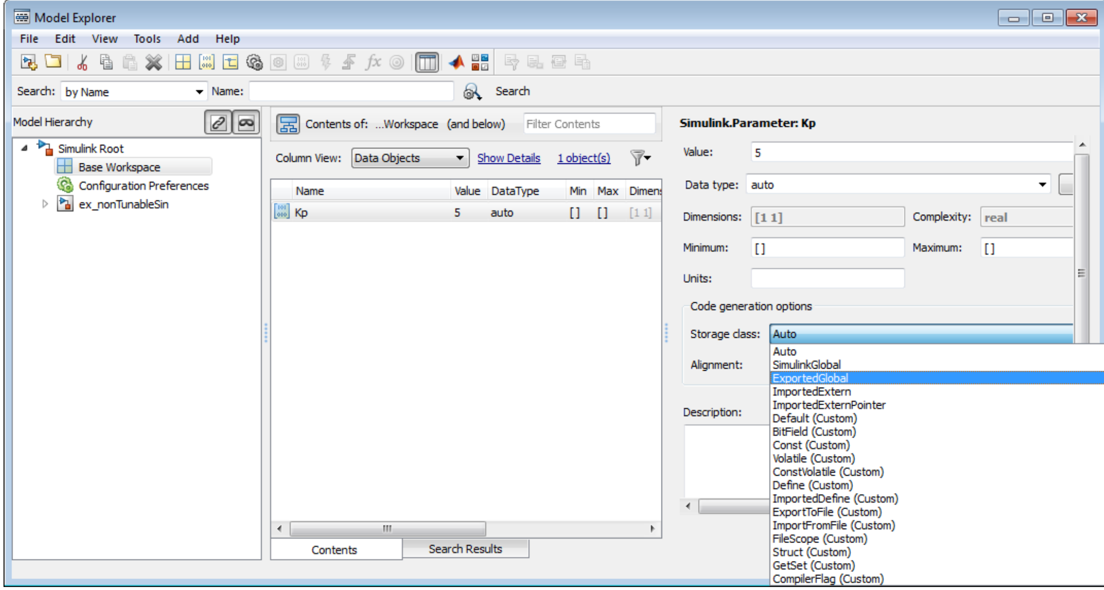
  a Select the **Value** field at the top of the dialog pane.

  b Type `5.0`.

  c Click the **Apply** button.

6 To set the `Kp.CoderInfo.StorageClass` in Model Explorer:

  a Click the **Storage class** menu and select `ExportedGlobal`, as shown in the next figure.



b Click **Apply**.

The following table shows the variable declarations for `Kp` and the code generated for the Gain block in the model shown in the preceding model, with the **Inline parameters** and **Eliminate superfluous local variables (Expression folding)** check boxes selected (which includes the gain computation in the output computation). An example is shown for each possible setting of `CoderInfo.StorageClass`. Global structures include the model name (symbolized as *`model_`* or *`_model`*).

StorageClass Property	**Generated Variable Declaration and Code**
`Auto`	`*model*_Y.Out1 = rtb_u * 5.0;`

`SimulinkGlobal`	`struct _Parameters_*model* {`
	`  real_T Kp;`
	`}`
	`.`
	`.`
	`Parameters_*model* *model*_P = {`
	`  5.0`
	`};`

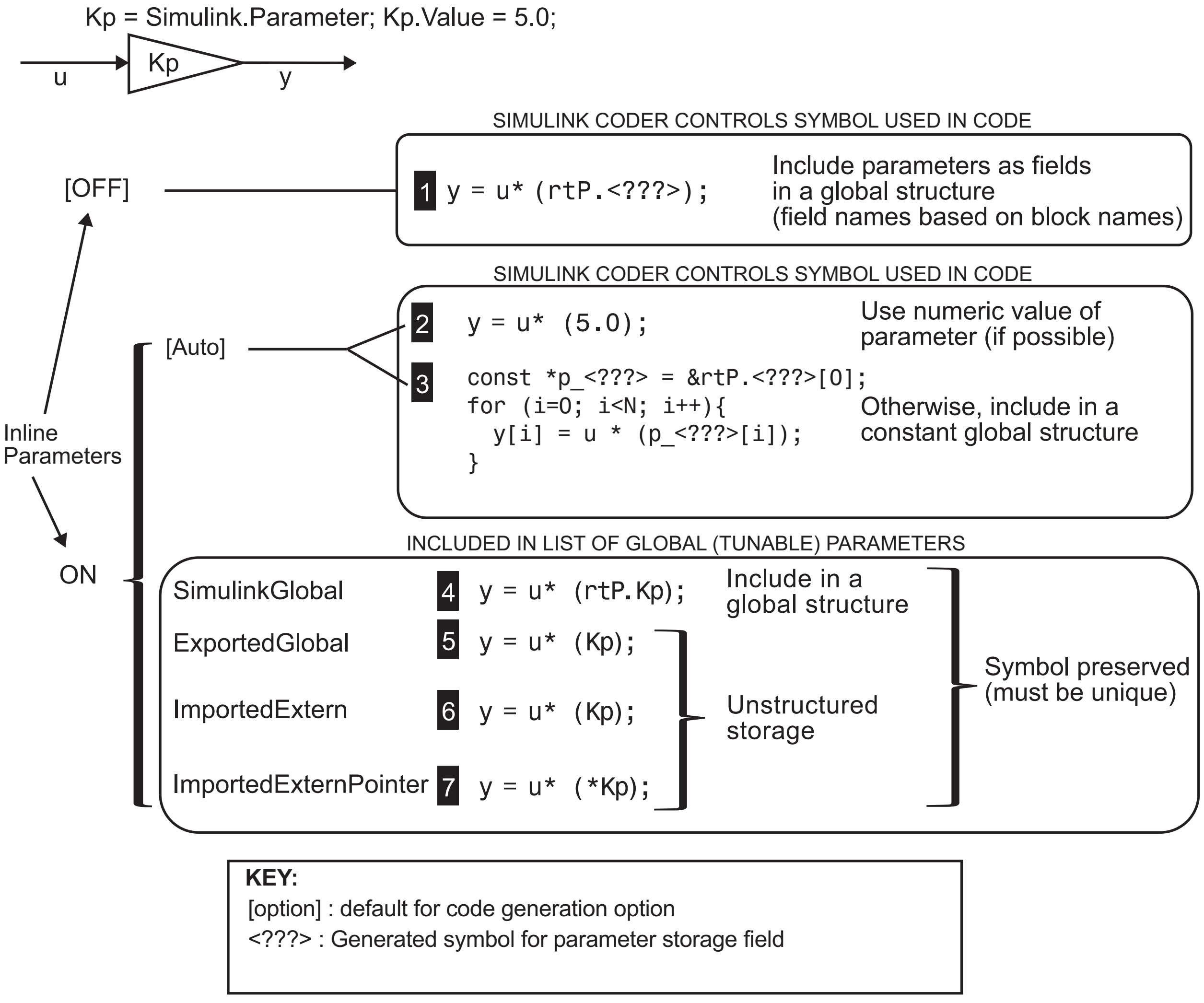
StorageClass Property	**Generated Variable Declaration and Code**
	. . *model*_Y.Out1 = rtb_u * *model*_P.Kp;

`ExportedGlobal`	extern real_T Kp;
	.
	.
	real_T Kp = 5.0;
	.
	.
	*model*_Y.Out1 = rtb_u * Kp;

`ImportedExtern`	extern real_T Kp; . . *model*_Y.Out1 = rtb_u * Kp;
`ImportedExternPointer`	extern real_T *Kp; . . *model*_Y.Out1 = rtb_u * (*Kp);

Parameter Object Configuration Quick Reference Diagram

The next figure shows the code generation and storage class options that control the representation of parameter objects in generated code.

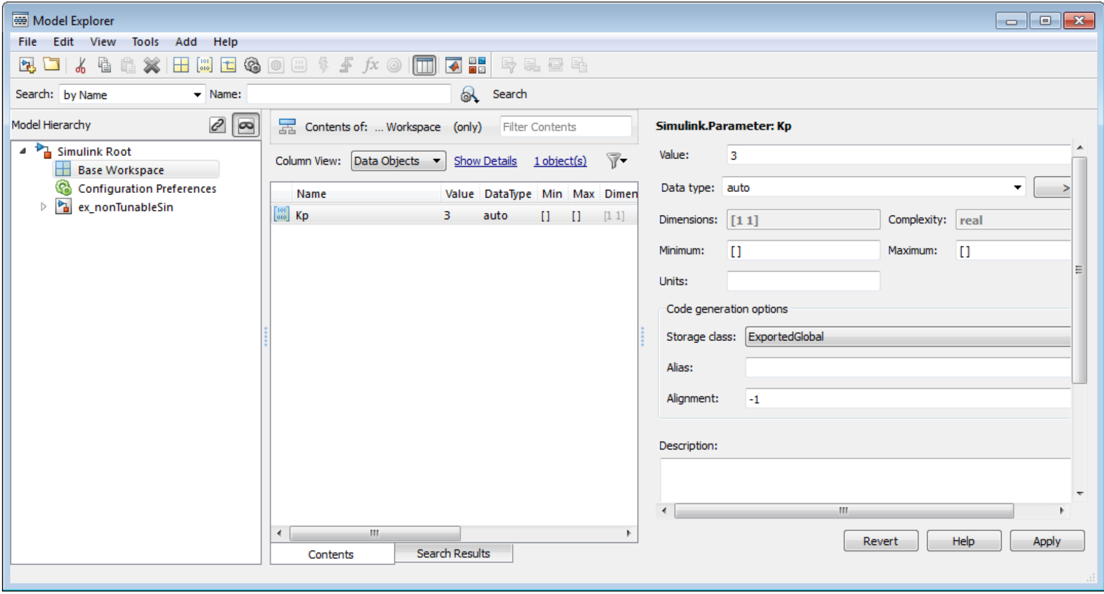


Resolve Conflicts in Parameter Object Configurations

Two methods are available for controlling the tunability of parameters. You can

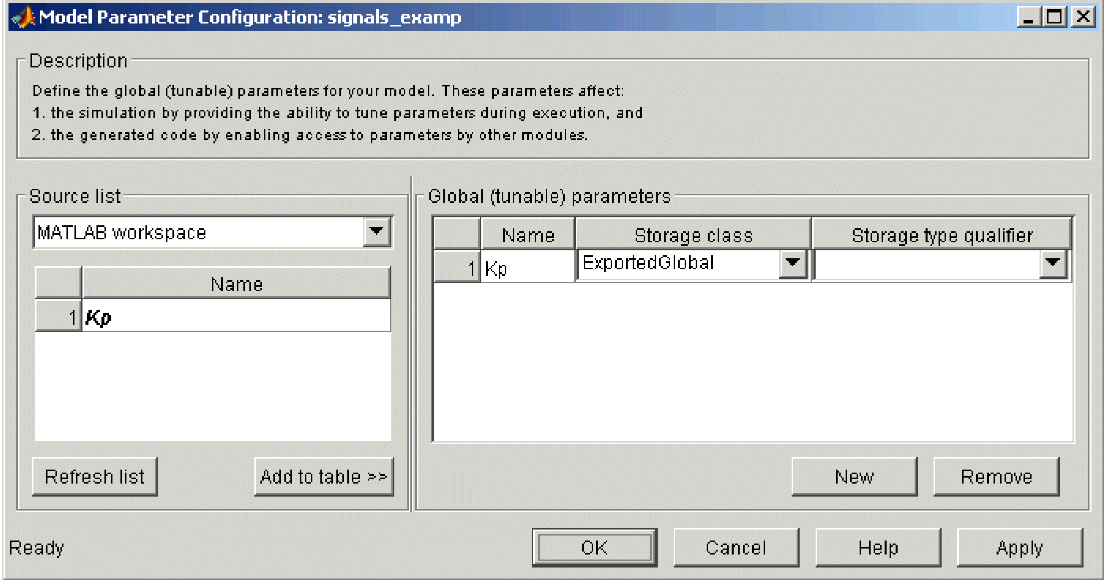
- Define them as `Simulink.Parameter` objects in the MATLAB workspace
- Use the Model Parameter Configuration dialog box

The next figures show how you can use each of these methods to control the tunability of parameter Kp. The first figure shows Kp defined as Simulink.Parameter in the Model Explorer. You control the tunability of Kp by specifying the parameter's storage class.



Parameter Object Kp with Auto Storage Class in Model Explorer

The next figure shows how you can use the Model Parameter Configuration dialog box to specify a storage class for numeric variables in the MATLAB workspace.



Parameter Kp Defined with SimulinkGlobal Storage Class

Note: Do not use both methods for controlling the tunability of a given parameter. If you use both methods and the storage class settings for the parameter do not match, an error results.

Signals

In this section...

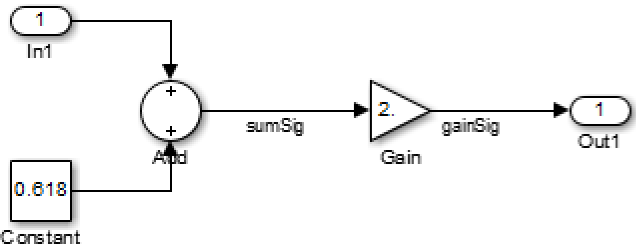
"About Signals" on page 8-46
"Signal Storage Concepts" on page 8-47
"Signals with Auto Storage Class" on page 8-49
"Signals with Test Points" on page 8-53
"Interface Signals to External Code" on page 8-53
"Symbolic Naming Conventions for Signals" on page 8-55
"Summary of Signal Storage Class Options" on page 8-56
"Interfaces for Monitoring Signals" on page 8-57
"Signal Objects" on page 8-57
"Initialize Signals and States Using Signal Objects" on page 8-65

About Signals

The Simulink Coder product offers a number of options that let you control how signals in your model are stored and represented in the generated code. This section discusses how you can use these options to

- 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 Simulink Coder product 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 in the Simulink Coder product

Global Block I/O Structure

By default, the Simulink Coder product 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 BlockIO_signal_examp signal_examp_B;

(in signal_examp.c)
/* Block signals (auto storage) */
BlockIO_signal_examp 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 8-55.

In certain cases, Simulink Coder will place 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.

Signals Storage Classes

In the Simulink Coder product, 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 Simulink Coder product, 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 block outputs**, **Enable local block outputs**, and **Minimize data copies between local and global variables** options, and on available stack space. See "Signals with Auto Storage Class" on page 8-49 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 `model_B`, or in unstructured global variables, independent of `model_B`. 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 block outputs**, **Eliminate superfluous local variables (expression folding)**, and **Minimize data

copies between local and global variables 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(Test Point)`: Test points are stored as fields of the *`model`*`_B` structure that are not shared or reused by another signal. See "Signals with Test Points" on page 8-53 for more information.
- `ExportedGlobal`: The signal is stored in a global variable, independent of the *`model`*`_B` data structure. *`model`*`.h` exports the variable. Signals with `ExportedGlobal` storage class must have unique signal names. See "Interface Signals to External Code" on page 8-53 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 "Interface Signals to External Code" on page 8-53 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 "Interface Signals to External Code" on page 8-53 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 block outputs**
- **Eliminate superfluous local variables (expression folding)**
- **Minimize data copies between local and global variables**

Use these options to control signal memory reuse and choose local or global (*`model`*`_B`) storage for signals. The **Signal storage reuse** option is on the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box, as shown in the next figure.

Simulation and code generation

Inline parameters Configure ... Signal storage reuse

Code generation

Enable local block outputs

Reuse block outputs

Eliminate superfluous local variables (expression folding)

Inline invariant signals

Minimize data copies between local and global variables

Use memcpy for vector assignment

Memcpy threshold (bytes): 64

Loop unrolling threshold: 5

Maximum stack size (bytes): Inherit from target

These options interact. When the **Signal storage reuse** option is selected,

- The **Reuse block outputs** option is enabled and selected, and signal memory is reused.
- The **Enable local block outputs** option is enabled and selected. This 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 **Minimize data copies between local and global variables** is enabled and cleared, and global memory is not reused.

The following code examples illustrate the effects of the **Signal storage reuse**, **Enable local block outputs**, **Reuse block outputs**, **Eliminate superfluous local variables (expression folding)** and **Minimize data copies between local and global variables** options. The examples were generated from the `signal_examp` model (see figure Signal_examp Model).

The first example illustrates signal storage optimization, with **Signal storage reuse**, **Enable local block outputs**, **Reuse block outputs**, and **Minimize data copies between local and global variables** selected. (For clarity in showing the individual Gain and Sum block computation, expression folding is off in this example.) The output signal from the Sum block reuses `signal_examp_Y.Out1`, a variable local to the model output function.

```
/* Model output function */
static void signal_examp_output(int_T tid)
```

```
{
  /* Sum: '<Root>Sum' incorporates:
   *  Constant: '<Root>/Constant'
   *  Inport: '<Root>>/In1'
   */
  signal_examp_Y.Out1 = signal_examp_U.In1 + signal_examp_P.Constant_Value;

  /* Gain: '<Root>/Gain' */
  signal_examp_Y.Out1 = signal_examp_P.Gain_Gain * signal_examp_Y.Out1;

  /* tid is required for a uniform function interface.
   * Argument tid is not used in the function. */
  UNUSED_PARAMETER(tid);
}
```

If you are constrained by limited stack space, you can turn **Enable local block outputs** off and still benefit from memory reuse. The following example was generated with **Enable local block outputs** cleared and **Signal storage reuse**, **Reuse block outputs**, and **Minimize data copies between local and global variables** selected. The output signals from the Sum and Gain blocks use global structure `signal_examp_B` rather than declaring local variables and in both cases the signal name is `gainSig`.

```
/* Model output function */
static void signal_examp_output(int_T tid)
{
  /* Sum: '<Root>/Add' incorporates:
   *  Constant: '<Root>/Constant'
   *  Inport: '<Root>/In1'
   */
  signal_examp_B.gainSig = signal_examp_U.In1 +
    signal_examp_P.Constant_Value;

  /* Gain: '<Root>/Gain' */
  signal_examp_B.gainSig = signal_examp_P.Gain_Gain *
    signal_examp_B.gainSig;

  /* Outport: '<Root>/Out1' */
  signal_examp_Y.Out1 = signal_examp_B.gainSig;

  /* tid is required for a uniform function interface.
   * Argument tid is not used in the function. */
  UNUSED_PARAMETER(tid);
}
```

When the **Signal storage reuse** option is cleared, **Reuse block outputs**, **Enable local block outputs**, and **Minimize data copies between local and global variables** are disabled. This makes the block output signals global and unique, `signal_examp_B.sumSig` and `signal_examp_B.gainSig`, as shown in the following code.

```
/* Model output function */
static void signal_examp_output(int_T tid)
```

```
{
  /* Sum: '<Root>/Add' incorporates:
   *  Constant: '<Root>/Constant'
   *  Inport: '<Root>/In1'
   */
  signal_examp_B.sumSig = signal_examp_U.In1 +
    signal_examp_P.Constant_Value;

  /* Gain: '<Root>/Gain' */
  signal_examp_B.gainSig = signal_examp_P.Gain_Gain *
    signal_examp_B.sumSig;

  /* Outport: '<Root>/Out1' */
  signal_examp_Y.Out1 = signal_examp_B.gainSig;

  /* tid is required for a uniform function interface.
   * Argument tid is not used in the function. */
  UNUSED_PARAMETER(tid);
}
```

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 block outputs ON	Signal storage reuse OFF (Reuse 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 *model*_B structure	Individual signal storage in *model*_B structure

Control Stack Space Allocation

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” for more information.

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 Simulink Coder build process allocates a separate memory buffer for each test point. Test points are stored as members of the `model_B` structure.

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 block outputs**
- **Eliminate superfluous local variables (expression folding)**
- **Minimize data copies between local and global variables**

For an example of storage declarations and code generated for a test point, see “Summary of Signal Storage Class Options” on page 8-56.

If you have an Embedded Coder license, you can specify that the Simulink Coder 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”.

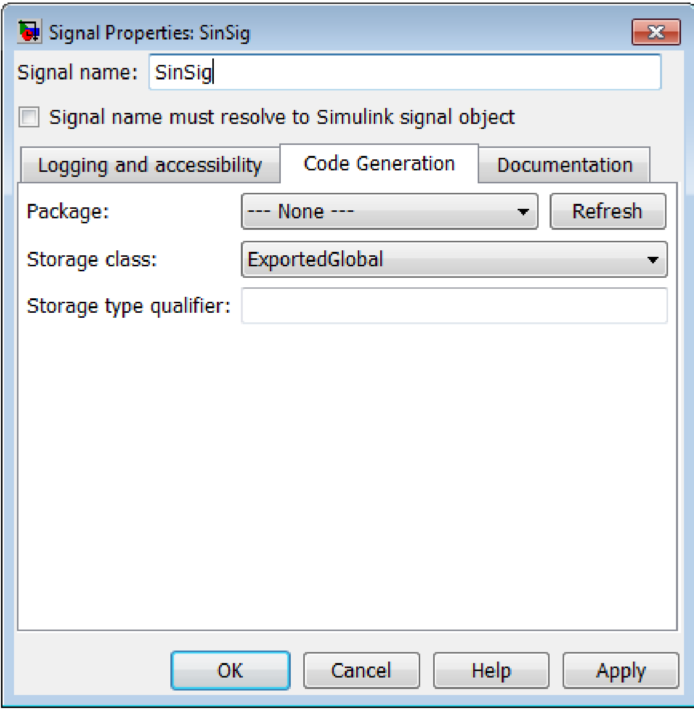
Interface Signals to External Code

The Simulink Signal Properties dialog box lets you interface selected signals to externally written code. In this way, your hand-written code has access to such signals for monitoring or other purposes. To interface a signal to external code, use the **Code Generation** tab of the Signal Properties dialog box to assign one of the following storage classes to the signal:

- `ExportedGlobal`
- `ImportedExtern`
- `ImportedExternPointer`

Set the storage class as follows:

1. In your Simulink block diagram, select the line that carries the signal. Then select **Signal Properties** from the **Edit** menu of your model. This opens the Signal Properties dialog box. Alternatively, right-click the line that carries the signal, and select **Signal properties** from the menu.
2. Select the **Code Generation** tab of the Signal Properties dialog box.
3. Select the desired storage class (`Auto`, `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer`) from the **Storage class** menu. The next figure shows `ExportedGlobal` selected.



4. *Optional*: For storage classes other than `Auto`, you can enter a storage type qualifier such as `const` or `volatile` in the **Storage type qualifier** field. The Simulink

Coder product does not check this string for errors; whatever you enter is included in the variable declaration.

5 Click **Apply**.

Note You can also interface test points and other signals that are stored as members of *model*_B to your code. To do this, your code must know the address of the *model*_B structure where the data is stored, and other information. This information is not automatically exported. The Simulink Coder product 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 8-57 for more information.

Symbolic Naming Conventions for Signals

When signals have a storage class other than `Auto`, the Simulink Coder product 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
- A unique *name mangling* string (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" for more detail.

When a signal has `Auto` storage class, the Simulink Coder 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 *model*.c or *model*.cpp
	
	`real_T rtb_sumSig;`
		`rtb_sumSig = signal_examp_U.In1 +`
		`  signal_examp_P.Constant_Value;`
		`rtb_sumSig *=`
		`  signal_examp_P.Gain_Gain;`
		`signal_examp_Y.Out1 = rtb_sumSig;`

Test point (for `sumSig` only)	In *model*.h
	
	`typedef struct`
	`_BlockIO_signal_examp`
	`{`
	`  real_T sumSig;`
	`}`
	`BlockIO_signal_examp;`
	
	In *model*.c or *model*.cpp
	
	`BlockIO_signal_examp`
	`signal_examp_B;`
	`real_T rtb_gainSig;`
		`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;`

`ExportedGlobal` (for `sumSig` only)	In *model*.h
	
	`extern real_T sumSig;`
	
	In *model*.c or *model*.cpp
	
	`real_T sumSig;`
	`real_T rtb_gainSig;`
		`sumSig = signal_examp_U.In1 +`
		`signal_examp_P.Constant_Value;`
		`rtb_gainSig = sumSig *`
		`  signal_examp_P.Gain_Gain;`
		`signal_examp_Y.Out1 = rtb_gainSig;`

`ImportedExtern`	In *model*_private.h
	
	`extern real_T sumSig;`
		`sumSig = signal_examp_U.In1 +`
		`signal_examp_P.Constant_Value;`
		`rtb_gainSig = sumSig *`
		`  signal_examp_P.Gain_Gain;`

Storage Class	Declaration	Code
	In *model*.c or *model*.cpp `real_T rtb_gainSig;`	`signal_examp_Y.Out1 = rtb_gainSig;`

ImportedExternPoi	In *model*_private.h
	`extern real_T *sumSig;`
	In *model*.c or *model*.cpp
	`real_T rtb_gainSig;`
		`(*sumSig) = signal_examp_U.In1 +`
		`signal_examp_P.Constant_Value;`
		`rtb_gainSig = (*sumSig) *`
		`signal_examp_P.Gain_Gain;`
		`signal_examp_Y.Out1 = rtb_gainSig;`

Interfaces for Monitoring Signals

The Simulink Coder product 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 “Data Interchange Using the C API” for information.
- An interface for exporting ASAP2 files, which you customize to use signal objects. For details, see “ASAP2 Data Measurement and Calibration”.

Signal Objects

- “About Signal Objects for Code Generation” on page 8-58
- “Use Signal Objects for Code Generation” on page 8-58
- “Configure Signal Objects for Code Generation” on page 8-58
- “Storage Classes for Signal Objects” on page 8-59
- “Configure Signal Objects from Command Line” on page 8-59
- “Configure Signal Objects Using Model Explorer” on page 8-61
- “Resolve Conflicts in Configuration of Signals Objects” on page 8-64

This section discusses how to use signal objects in code generation. Signal objects can be used to represent both signal and state data, and behave similarly to parameter objects, described in “Parameter Objects” on page 8-35.

About Signal Objects for Code Generation

Within the class hierarchy of Simulink data objects, the Simulink product provides a class that is designed as base class for signal storage. This topic explains how to use signal objects in code generation.

The `CoderInfo` properties of signal objects are used by the Simulink Coder product during code generation. These properties let you assign storage classes to the objects, thereby controlling how the generated code stores and represents signals.

The Simulink Coder build process also writes information about the properties of signal objects to the *`model`*`.rtw` file. This information, formatted as `Object` records, is accessible to Target Language Compiler programs. For general information on `Object` records, see “Data Object Information in model.rtw”.

Before using Simulink signal objects with the Simulink Coder product, read the discussion of Simulink data objects in the Simulink documentation.

Use Signal Objects for Code Generation

The general procedure for using signal objects in code generation is as follows:

1. Define a subclass of `Simulink.Signal`.
2. Instantiate signal objects from your subclass and set their properties from the command line or by using Model Explorer.
3. Use the objects as signals within your model.
4. Generate code and build your target executable.

Configure Signal Objects for Code Generation

In configuring signal objects for code generation, you use the following code generation options and signal object properties:

- The **Signal storage reuse** code generation option (see “Signals” on page 8-46).
- The **Enable local block outputs** code generation option (see “Signals” on page 8-46).
- The **Minimize data copies between local and global variables** code generation option (see “Signals” on page 8-46).
- The `CoderInfo.StorageClass` signal object property: The storage classes defined for signal objects, and their effect on code generation, are the same for model signals and signal objects (see “Signals Storage Classes” on page 8-48).

Other signal object properties (such as user-defined properties of classes derived from `Simulink.Signal`) do not affect code generation.

Storage Classes for Signal Objects

The way in which the Simulink Coder product uses storage classes to determine how signals are stored is the same with and without signal objects. However, if a signal's label resolves to a signal object, the object's `CoderInfo.StorageClass` property is used in place of the port configuration of the signal.

The default storage class is `Auto`. If the storage type is `Auto`, the Simulink Coder product follows the **Signal storage reuse**, **Reuse block outputs**, **Enable local block outputs**, **Eliminate superfluous local variables (expression folding)**, and **Minimize data copies between local and global variables** code generation options to determine whether signal objects are stored in reusable and/or local variables. Make sure that these options are set for your application.

To generate a test point or signal storage declaration that can interface externally, use an explicit `CoderInfo.StorageClass` assignment. For example, setting the storage class to `SimulinkGlobal`, as in the following command, is equivalent to declaring a signal as a test point.

```
SinSig.CoderInfo.StorageClass = 'SimulinkGlobal';
```

Configure Signal Objects from Command Line

The discussion and code examples in this section refer to the model shown in the next figure.

To configure a signal object, you must first create it and associate it with a labeled signal in your model. To do this,

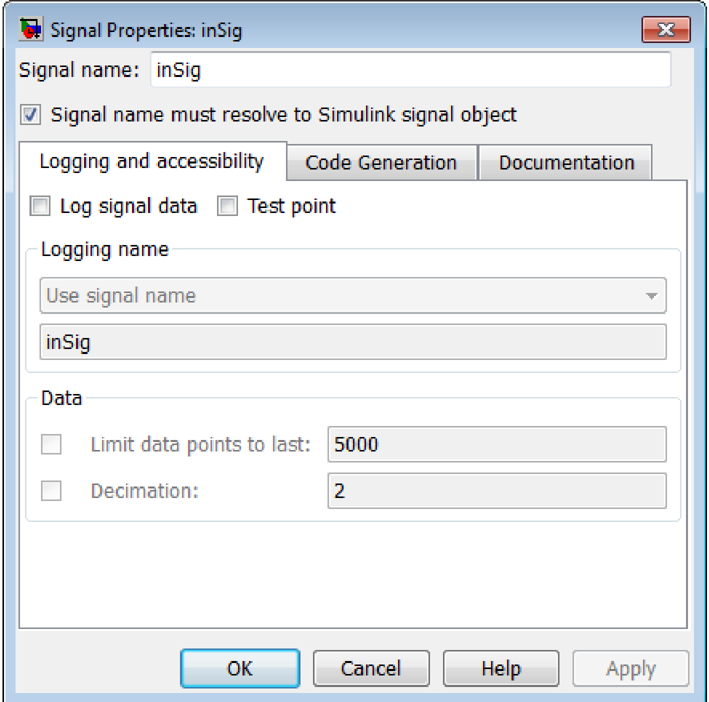
1. Define a subclass of `Simulink.Signal`. In this example, the signal object is an instance of the class `Simulink.Signal`, which is provided with the Simulink product.

2 Instantiate a signal object from your subclass. The following example instantiates `inSig`, a signal object of class `Simulink.Signal`.

```
inSig = Simulink.Signal
inSig =
Simulink.Signal
         CoderInfo: [1x1 Simulink.SignalCoderInfo]
     Description: ''
        DataType: 'auto'
             Min: []
             Max: []
        DocUnits: ''
      Dimensions: -1
      Complexity: 'auto'
      SampleTime: -1
    SamplingMode: 'auto'
    InitialValue: ''
```

Make sure that the name of the signal object matches the label of the desired signal in your model. This enables the Simulink engine to resolve the signal label to the corresponding object. For example, in the model shown in the above figure, the signal label `inSig` would resolve to the signal object `inSig`.

3 You can require signals in a model to resolve to `Simulink.Signal` objects. To do this for the signal `inSig`, in the model window right-click the signal line labeled `inSig` and choose **Signal Properties** from the context menu. A Signal Properties dialog appears.



4 In the Signal Properties dialog box that appears, select the check box labelled **Signal name must resolve to Simulink signal object**, and click **OK** or **Apply**.

5 Set the object properties as required. You can do this by using the Simulink Model Explorer. Alternatively, you can assign properties by using MATLAB commands. For example, assign the signal object's storage class by setting the `CoderInfo.StorageClass` property as follows.

```
inSig.CoderInfo.StorageClass = 'ExportedGlobal';
```

Configure Signal Objects Using Model Explorer

If you prefer, you can create signal objects and modify their attributes using Model Explorer. This lets you see and set attributes of a signal in a dialog box pane, and alleviates the need to remember and type field names. Do the following to instantiate `inSig` and set its attributes from Model Explorer:

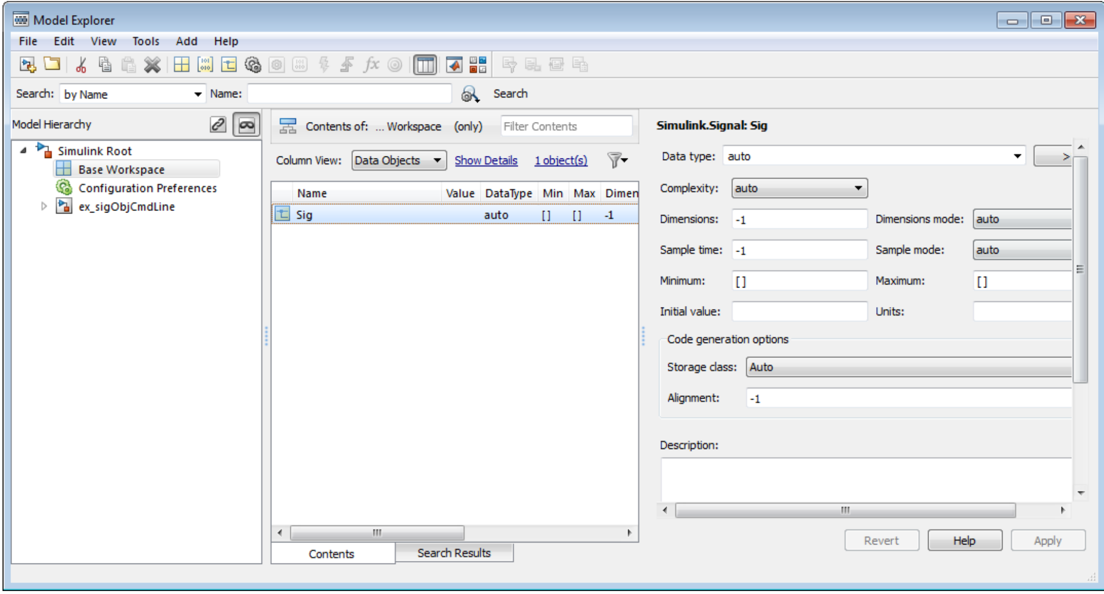
1 Choose **Model Explorer** from the View **menu**.

Model Explorer opens or activates if it already was open.

2 Select `Base Workspace` in the **Model Hierarchy** pane.

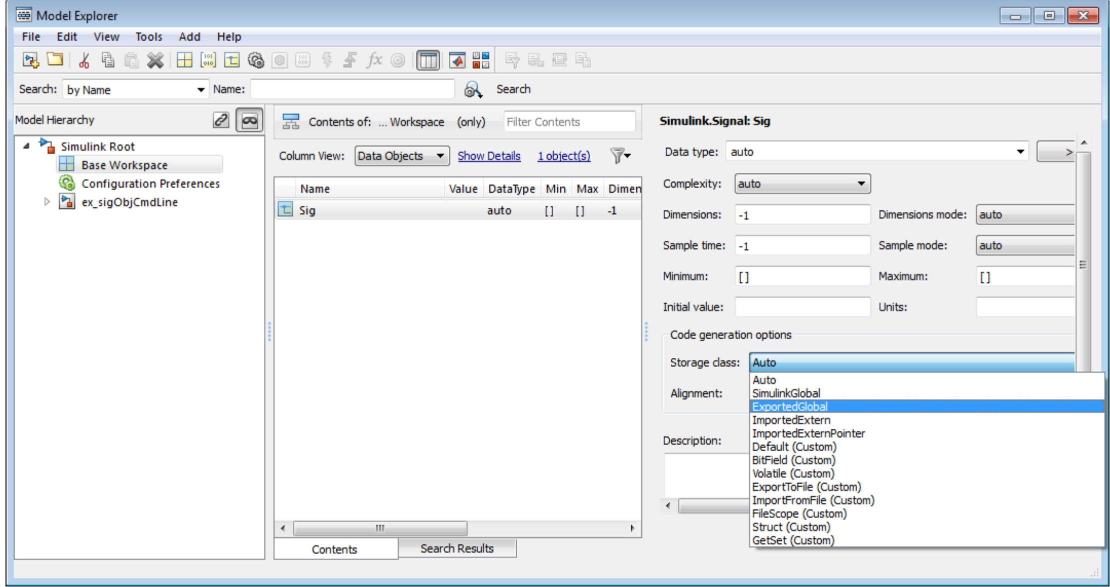
3 Select **Simulink Signal** from the **Add** menu.

A new signal named `Sig` appears in the **Contents** pane.



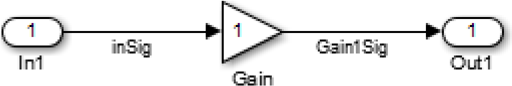
4 To set the signal name in Model Explorer, click the word `Sig` in the **Name** column to select it, and rename it by typing `inSig` followed by **Return** in place of `Sig`.

5 To set the `inSig.CoderInfo.StorageClass` in Model Explorer, click the **Storage class** menu and select `ExportedGlobal`, as shown in the next figure.



6 Click **Apply**.

The following table shows, for each setting of `CoderInfo.StorageClass`, the variable declaration and the code generated for the inport signal (`inSig`) of the current model:



Storage Class	Declaration	Code

`Auto` (with storage optimizations on)	In *`model`*`.h`
	
	`typedef struct`
	`_ExternalInputs_signal_ objs_examp_tag`
	`{`
	`  real_T inSig;`
	` }`
	`ExternalInputs_signal_ objs_examp;`
		`rtb_Gain1Sig =`
		`  signal_objs_examp_U.inSig *`
		`  signal_objs_examp_P.Gain_Gain;`

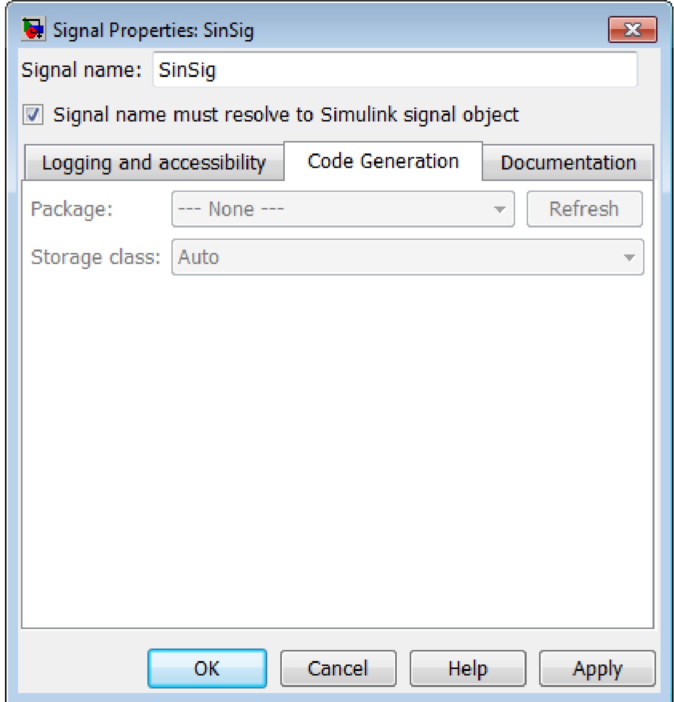
Storage Class	Declaration	Code

SimulinkGlobal	In *model*.h
	
	typedef struct
	_ExternalInputs_signal_objs_examp_tag
	{
	  real_T inSig;
	 }
	ExternalInputs_signal_objs_examp;
		rtb_Gain1Sig =
		  signal_objs_examp_U.inSig *
		  signal_objs_examp_P.Gain_Gain;

ExportedGlobal	In *model*.c or *model*.cpp real_T inSig; In *model*.h extern real_T inSig;	rtb_Gain1Sig = inSig * signal_objs_examp_P.Gain_Gain;
ImportedExtern	In *model*_private.h extern real_T inSig;	rtb_Gain1Sig = inSig * signal_objs_examp_P.Gain_Gain;
ImportedExtern	In *model*_private.h extern real_T *inSig;	rtb_Gain1Sig = (*inSig) * signal_objs_examp_P.Gain_Gain;

Resolve Conflicts in Configuration of Signals 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 must resolve to a Simulink data object. To do this, select the **Signal name must resolve to Simulink signal object** option in the Signal Properties dialog box. When you do this, you cannot specify the **Storage class** property in the **Code Generation** tab of the Signal Properties dialog box, as the next figure shows.



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.

Initialize Signals and States Using Signal Objects

You can use Simulink signal objects to initialize signals and discrete states with user-defined values for simulation and code generation. 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 details on simulation behavior, see “Initialization Behavior Summary for Signal Objects” in the Simulink documentation.

Specify Initial Value for Signal Object

You can use signal objects that have a storage class other than `'auto'` or `'SimulinkGlobal'` to initialize

- Discrete states with an initial condition parameter
- Signals in a model except bus signals and signals with constant sample time

The initial value is the signal or state value before a simulation takes its first time step.

Note: Some initial value settings may depend on the initialization mode. For more information, see “Underspecified initialization detection”.

Classic initialization mode: In this mode, initial value settings for signal objects that represent the following signals and states override the corresponding block parameter initial values if undefined (specified as `[]`):

- Output signals of conditionally executed subsystems and Merge blocks
- Block states

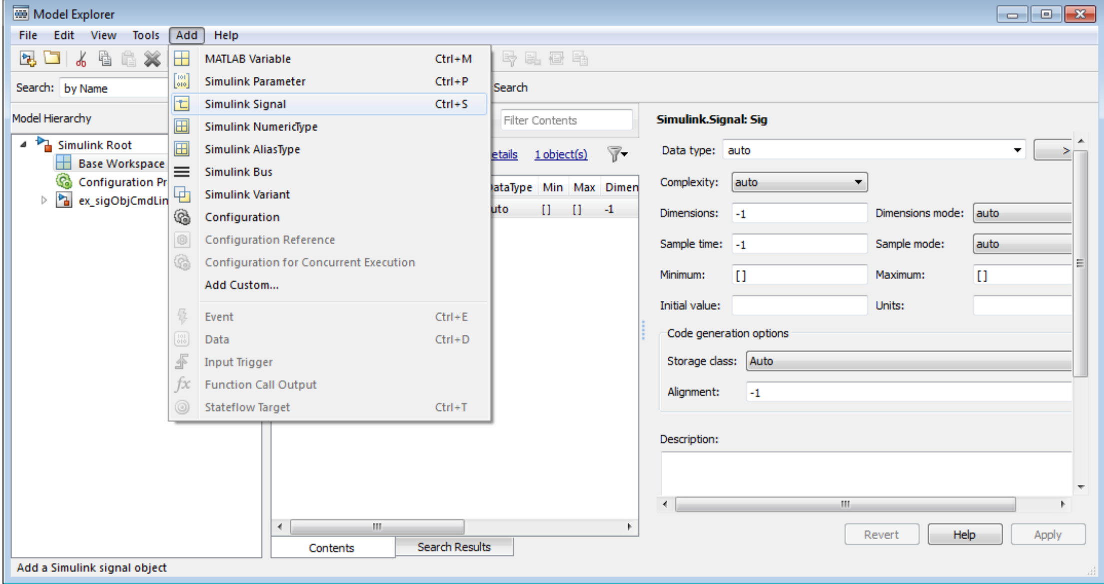
Simplified initialization mode: In this mode, initial values of signal objects associated with the output of the following blocks are ignored. The initial values of the corresponding blocks are used instead.

- Output signals of conditionally executed subsystems
- Merge blocks

To specify an initial value, use the Model Explorer or MATLAB commands to do the following:

1. Create the signal object.

   Model Explorer



MATLAB Command

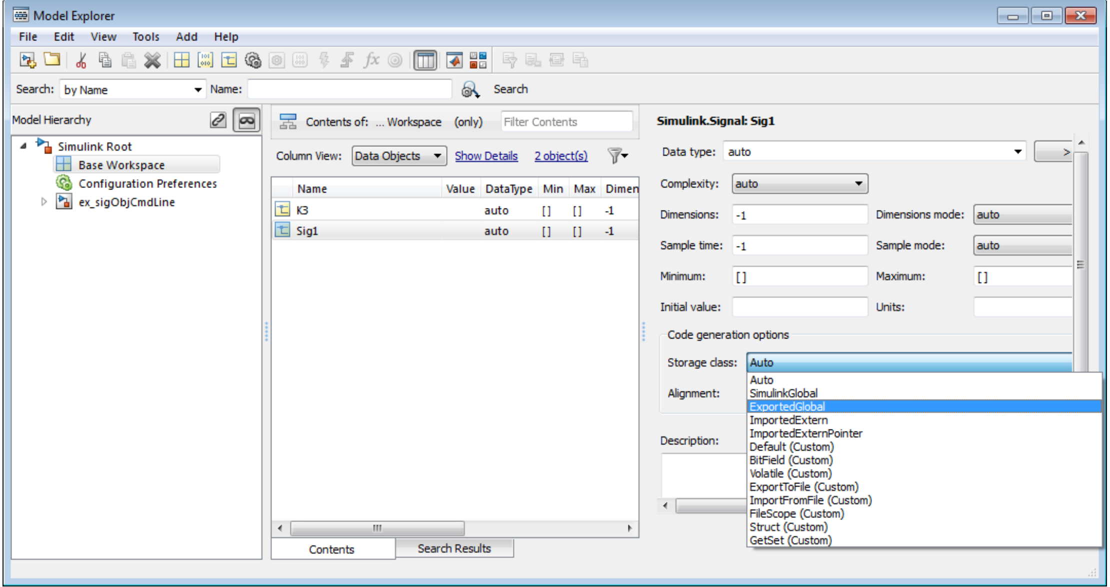
```
S1=Simulink.Signal;
```

The name of the signal object must be the same as the name of the signal that the object is initializing. Although not required, consider setting the **Signal name must resolve to Simulink signal object** option in the Signal Properties dialog box. This setting makes signal objects in the MATLAB workspace consistent with signals that appear in your model.

Consider using the Data Object Wizard to create signal objects. The Data Object Wizard searches a model for signals for which signal objects do not exist. You can then selectively create signal objects for multiple signals listed in the search results with a single operation. For more information about the Data Object Wizard, see “Data Object Wizard” in the Simulink documentation.

2. Set the signal object's storage class to a value other than `'auto'` or `'SimulinkGlobal'`.

Model Explorer



MATLAB Command

```
S1.CoderInfo.StorageClass='ExportedGlobal';
```

3 Set the initial value. You can specify a MATLAB string expression that evaluates to a `double` numeric scalar value or array.

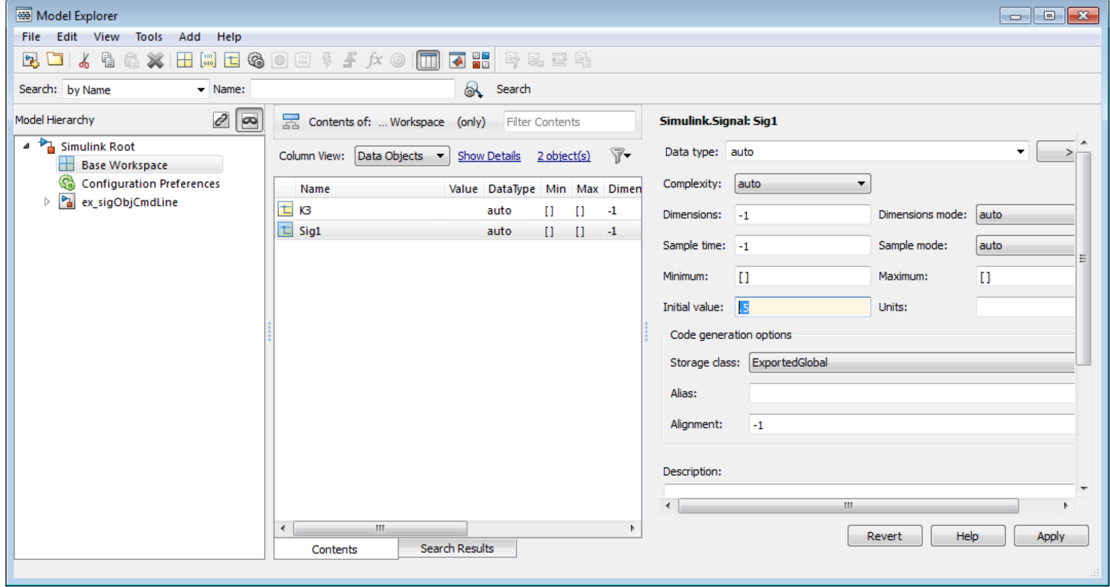
	Model Explorer	MATLAB Command

Valid	`1.5`
	`[1 2 3]`
	`1+0.5`
		`foo = 1.5;`
		`s1.InitialValue = 'foo';`

Invalid	`uint(1)`	`foo = '1.5';` `s1.InitialValue = 'foo';`

The Simulink engine converts the initial value so the type, complexity, and dimension are consistent with the corresponding block parameter value. If you specify an invalid value or expression, an error message appears when you update the model.

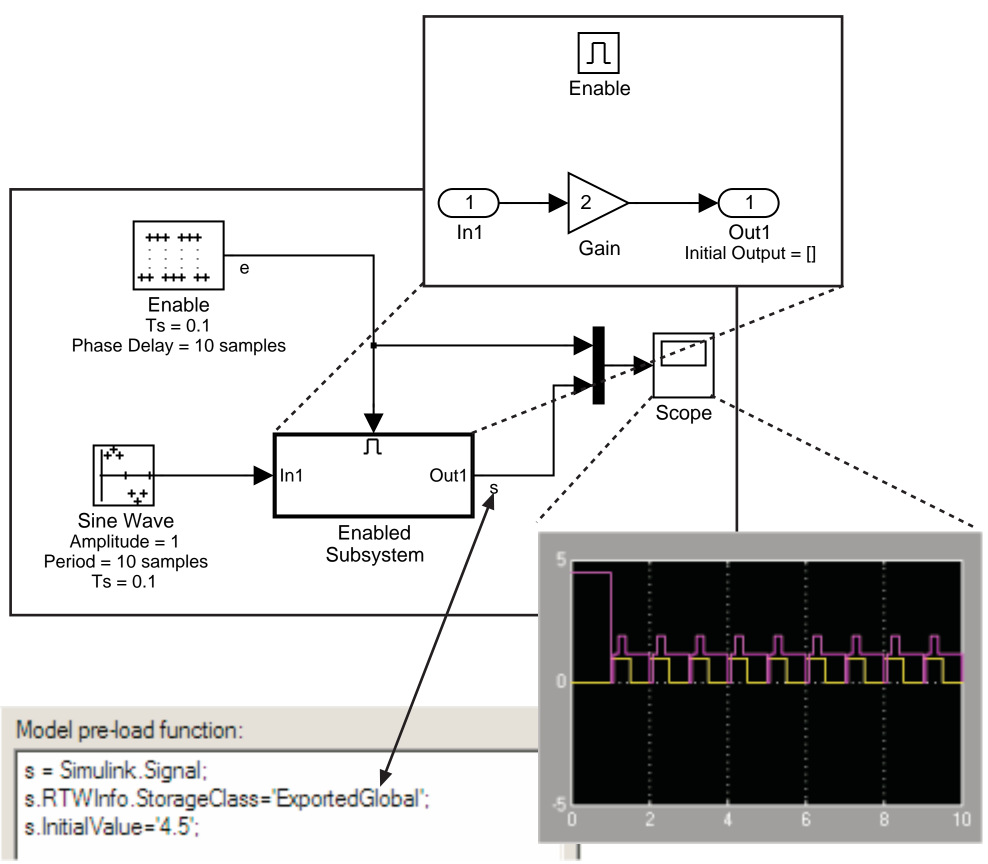
Model Explorer



MATLAB Command

```
S1.InitialValue='0.5'
```

The following example shows a signal object specifying the initial output of an enabled subsystem.



Signal s is initialized to 4.5. Note that to avoid a consistency error, the initial value of the enabled subsystem's Outport block must be [ ] or 4.5.

Signal Object Initialization in Generated Code

The initialization behavior for code generation is the same as that for model simulation with the following exceptions:

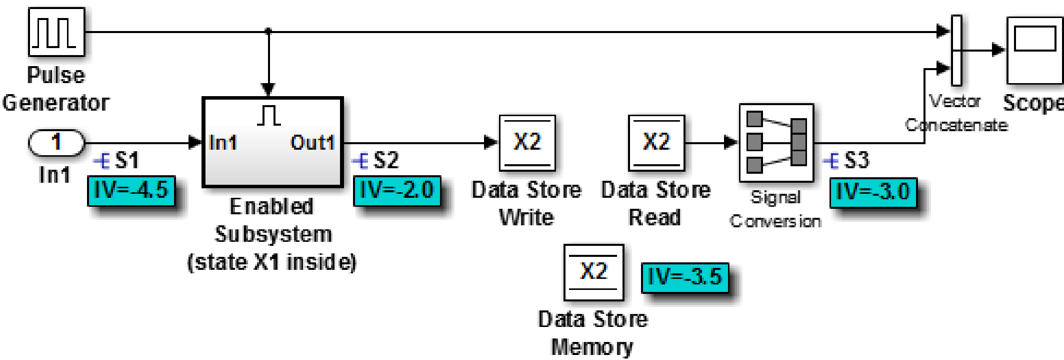
- RSim executables can use the **Data Import/Export** pane of the Configuration Parameters dialog box to load input values from MAT-files. GRT and ERT executables cannot load input values from MAT-files.
- The initial value for a block output signal or root level input or output signal can be overwritten 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.

For details on initialization behavior for different types of signals and discrete states, see "Initialization Behavior Summary for Signal Objects" in the Simulink documentation.

When you initialize Simulink signal objects in a model during code generation, the corresponding 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 Simulink Coder product 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;

}
```

Also note that for an enabled subsystem, such as the one shown in the preceding model, 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 `rtwdeni_sigobj_iv_output`.

```
/* Model output function */

static void rtwdemo_sigobj_iv_output(void)
{
         .
         .
         .
 /* Disable for enable system: '<Root>/Enabled Subsystem (state X1 inside)' */

    /* (Virtual) Outport Block: '<S2>/Out1' */

    S2 = aa2;
```

Tunable Initial Values

If you specify a tunable parameter in the initial value for a signal object, the parameter expression is preserved in the initialization code in *`model`*`.c`.

For example, if you configure parameter `df` to be tunable for model `signal_iv` and you initialize the signal object for discrete state `X1` with the expression `df*2`, the following initialization code appears for signal object `X1` in `signal_iv.c`.

```
void MdlInitialize(void) {
```

```
/* InitializeConditions for UnitDelay: '<Root>/Unit Delay X1=2' */
X1 = (tunable_param_P.df * 2.0);
}
```

For more information about the treatment of tunable parameters in generated code, see “Parameters” on page 8-11.

States

In this section...

"About States" on page 8-74
"Symbolic Names for Continuous Block States" on page 8-74
"State Attributes" on page 8-76
"State Storage" on page 8-76
"State Storage Classes" on page 8-77
"Interface States to External Code" on page 8-78
"Symbolic Names for States" on page 8-80
"Control Code Generation for Block States" on page 8-83
"Summary of State Storage Class Options" on page 8-83

About States

For certain block types, you can assign names to block states in generated code using Simulink Coder.

For certain discrete block types, you can also control how block states in your model are represented in the generated code. Using the **State Attributes** tab in a block dialog box, you can specify whether states declared in generated code are interfaceable (visible) to externally written code. You can also specify that states be stored in locations declared by externally written code. For more information, see "State Attributes" on page 8-76 .

Symbolic Names for Continuous Block States

The following information applies to these continuous blocks. To name states for other types of blocks, see "Symbolic Names for States" on page 8-80.

- Integrator
- Integrator Limited
- Integrator Second Order
- Integrator Limited Order
- PID Controller

- PID Controller (2DOF)
- State Space
- Transfer Fcn
- Variable Time Delay
- Variable Transport Delay
- Zero-Pole

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.
- On the **Function Block Parameters** pane, in the **State name** field, specify a name.

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 string 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

On the **Function Block Parameters** pane, 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 **Function Block Parameters** pane, in the **State Name** field, enter the string that you want to use for the state name. Use single quotes around the state name.
3. Click **OK** or **Apply** to accept the value.
4. Generate code and examine it to see the new identifier name.

State Attributes

For certain 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:

- 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.
- Assign symbolic names to block states in generated code.

State Storage

The following information applies to these discrete blocks:

- Discrete Filter
- Discrete PID Controller
- Discrete PID Controller (2DOF)
- Discrete State-Space
- Discrete-Time Integrator
- Discrete Transfer Function
- Discrete Zero-Pole
- Memory
- Unit 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 hand-written code, you can specify that the state is to be stored in a location other than the `DWork` vector. You do this by assigning a storage class to the block state.

You can also define a symbolic name, to be used in code generation, for a block state.

State Storage Classes

The storage class property of a block state specifies how the Simulink Coder product declares and stores the state in a variable. Storage class options for block states are similar to those for signals. The available storage classes are

- `Auto`
- `ExportedGlobal`
- `ImportedExtern`
- `ImportedExternPointer`

Default Storage Class

`Auto` is the default storage class and is the storage class you should use for states that you do not need to interface to external code. States with `Auto` storage class are stored as members of the `Dwork` vector.

You can assign a symbolic name to states with `Auto` storage class. If you do not supply a name, the Simulink Coder product generates one, as described in “Symbolic Names for States” on page 8-80.

Explicitly Assigned Storage Classes

Block states with storage classes other than `Auto` are stored in unstructured global variables, independent of the `Dwork` vector. These storage classes are for states that you want to interface to external code. The following storage classes are available for states:

- `ExportedGlobal`: The state is stored in a global variable. `model.h` exports the variable. States with `ExportedGlobal` storage class must have unique names.
- `ImportedExtern`: `model_private.h` declares the state as an `extern` variable. Your code must supply the variable definition. States with `ImportedExtern` storage class must have unique names.

- `ImportedExternPointer`: *`model`*`_private.h` declares the state as an `extern` pointer. Your code must supply the pointer variable definition. States with `ImportedExternPointer` storage class must have unique names.

The table in "Summary of Signal Storage Class Options" on page 8-56 gives examples of variable declarations and the code generated for block states with each type of storage class.

Note: Assign a symbolic name to states to specify a storage class other than `auto`. If you do not supply a name for `auto` states, the Simulink Coder product generates one, as described in "Symbolic Names for States" on page 8-80.

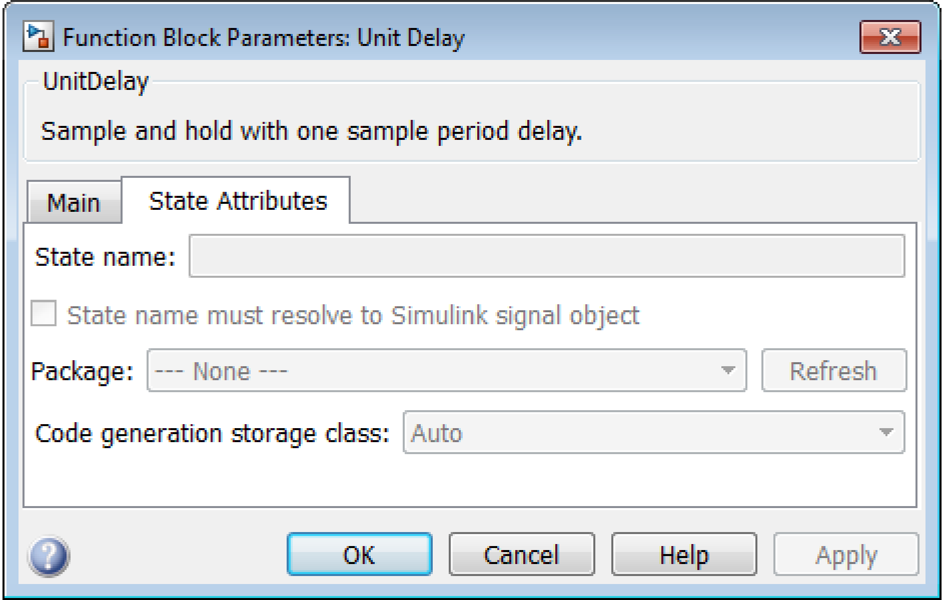
The next section explains how to use the **State Attributes** tab of the block dialog box to assign storage classes to block states.

Interface States to External Code

In the **State Attributes** tab of a block parameter dialog box, you can interface a block's state to external code by assigning the state a storage class other than `Auto` (that is, `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer`).

Set the storage class as follows:

1. In your block diagram, double-click the desired block. This action opens the block dialog box with two or more tabs, which includes **State Attributes**.
2. Click the **State Attributes** tab.



3 Enter a name for the variable to be used to store block state in the **State name** field.

The **State name** field turns yellow to indicate that you changed it.

4 Click **Apply** to register the variable name.

The first two fields beneath the **State name**, **State name must resolve to Simulink signal object** and **Code generation storage class**, become enabled.

5 If the state is to be stored in a Simulink signal object in the base or model workspace, select **State name must resolve to Simulink signal object**.

If you choose this option, you cannot declare a storage class for the state in the block, and the fields below become disabled.

6 Select the desired storage class (`ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer`) from the **Code generation storage class** menu.

7 *Optional*: For storage classes other than `Auto`, you can enter a storage type qualifier such as `const` or `volatile` in the **Code generation storage type qualifier** field. The Simulink Coder product does not check this string for errors; what you enter is included in the variable declaration.

8 Click **OK** or **Apply** and close the dialog box.

Symbolic Names for States

To specify names for other block states, see “Symbolic Names for Continuous Block States” on page 8-74.

To determine the variable or field name generated for a block's state, you can:

- Use a default name generated by the Simulink Coder product
- Define a symbolic name by using the **State name** field of the **State Attributes** tab in a block dialog box

Default Block State Naming Convention

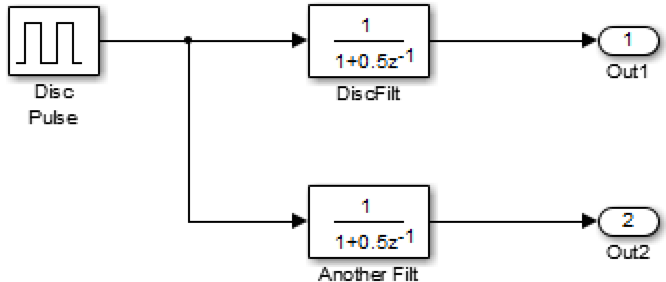
If you do not define a symbolic name for a block state, the Simulink Coder product uses the following default naming convention:

```
BlockType#_DSTATE
```

where

- `BlockType` is the name of the block type (for example, Discrete_Filter).
- # is a unique ID number (#) assigned by the Simulink Coder product if multiple instances of the same block type appear in the model. The ID number is appended to `BlockType`.
- `_DSTATE` is a string 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 block's 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.

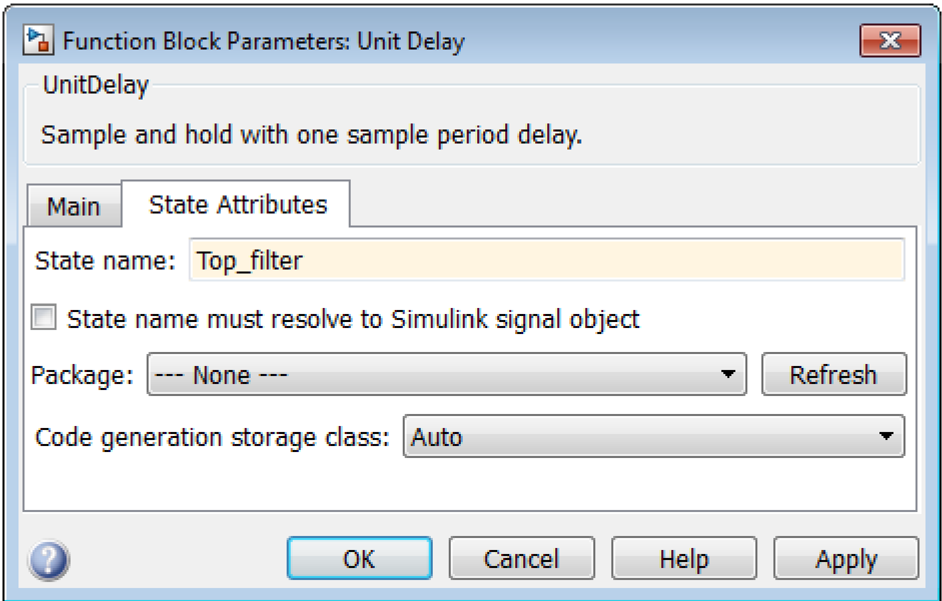
The states of the two Discrete Filter blocks are stored in `DWork` vectors, initialized as shown in the code below:

```
/* 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 tabs, 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**. The dialog box now looks like this:



5 Click **OK**.

The following state initialization code was generated from the example model shown in "Configure Signal Objects from Command Line" on page 8-59, 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;
```

Control Code Generation for Block States

If you are not familiar with Simulink data objects and signal objects, you should read “Signals” on page 8-46 before reading this section.

You can associate a block state with a signal object and control code generation for the block state through the signal object:

1. Instantiate the desired signal object, and set its `CoderInfo.StorageClass` property.
2. Open the dialog box for the block whose state you want to associate with the signal object.
3. Click the **State Attributes** tab.
4. Enter the name of the signal object in the **State name** field.
5. Select **State name must resolve to Simulink signal object**.

   This step disables the **Code generation storage class** and **Code generation storage type qualifier** options in the **State Attributes** tab, because the signal object specifies these settings.
6. Click **Apply** and close the dialog box.

Note: When a block state is associated with a signal object, the mapping between the block state and the signal object must be one-to-one. If two or more identically named entities, such as a block state and a signal, map to the same signal object, the name conflict is flagged as an error at code generation time.

Summary of State Storage Class Options

Here is a simple model, `unit_delay`, which contains a Unit Delay block:

The following table shows, for each state storage class option, the variable declaration and initialization code generated for a Unit Delay block state. The block state has the user-defined state name `udx`.

Storage Class	Declaration	Initialization Code

`Auto`	In *model*.h
	
	`typedef struct`
	`D_Work_unit_delay_tag`
	`{`
	`  real_T udx;`
	` }`
	`D_Work_unit_delay;`
		`unit_delay_DWork.udx = 0.0;`

`Exported Global`	In *model*.c or *model*.cpp
	
	`real_T udx;`
	
	In *model*.h
	
	`extern real_T udx;`
		In *model*.c or *model*.cpp
		
		`udx = 0.0;`

`ImportedExtern`	In *model*_private.h
	
	`extern real_T udx;`
		In *model*.c or *model*.cpp
		
		`udx =`
		`  unit_delay_P.UnitDelay_X0;`

`ImportedExternPointer`	In *model*_private.h
	
	`extern real_T *udx;`
		In *model*.c or *model*.cpp
		
		`(*udx) =`
		`  unit_delay_P.UnitDelay_X0;`

Data Stores

In this section...
"About Data Stores" on page 8-85
"Storage Classes for Data Store Memory Blocks" on page 8-85
"Generate Code for Data Store Memory Blocks" on page 8-87
"Nonscalar Data Stores in Generated Code" on page 8-88
"Data Store Buffering in Generated Code" on page 8-90

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.

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.

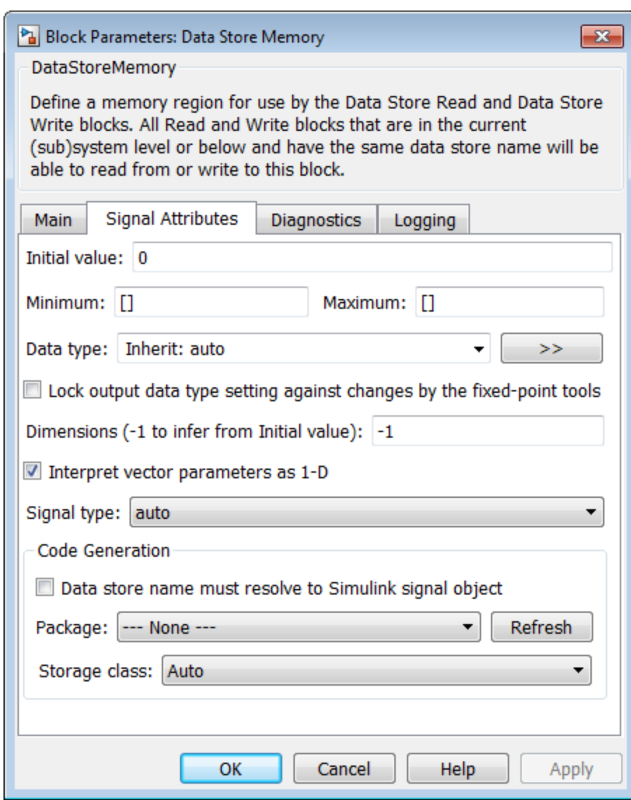
Data Store Memory blocks, like block states, have `Auto` storage class by default, and their memory is stored within the `DWork` vector. The symbolic name of the storage location is based on the data store name.

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 Simulink Coder product generates 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** > **Storage class** and **Code Generation** > **Storage type qualifier** fields of the Data Store Memory block dialog box. 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`	In *`model`*`.h`
	
	`typedef struct`
	`D_Work_tag`
	`{`
	`  real_T myData;`
	`}`
	`D_Work;`
	
	In *`model`*`.c` or *`model`*`.cpp`
	
	`/* Block states (auto storage) */`
	`D_Work` *`model`*`_DWork;`
		`model``_DWork.myData =`
		`  rtb_SineWave;`

`ExportedGlobal`	In *`model`*`.c` or *`model`*`.cpp`
	
	`/* Exported block states */`
	`real_T myData;`
	
	In *`model`*`.h`
	
	`extern real_T myData;`
		`myData = rtb_SineWave;`

`ImportedExtern`	In *`model`*`_private.h` `extern real_T myData;`	`myData = rtb_SineWave;`
`ImportedExternPointer`	In *`model`*`_private.h` `extern real_T *myData;`	`(*myData) = rtb_SineWave;`

Generate Code for Data Store Memory Blocks

If you are not familiar with Simulink data objects and signal objects, you should read "Signals" on page 8-46 before reading this section.

You can associate a Data Store Memory block with a signal object, and control code generation for the block through the signal object. To do this:

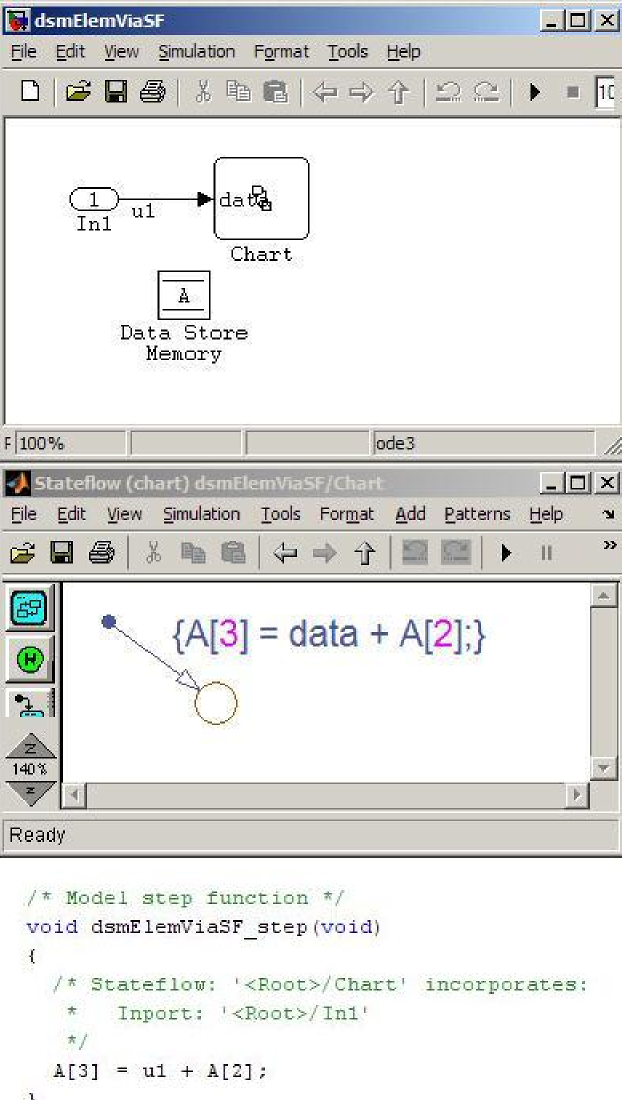
1. Instantiate the desired signal object.

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 Signals Objects" on page 8-64 for more information.

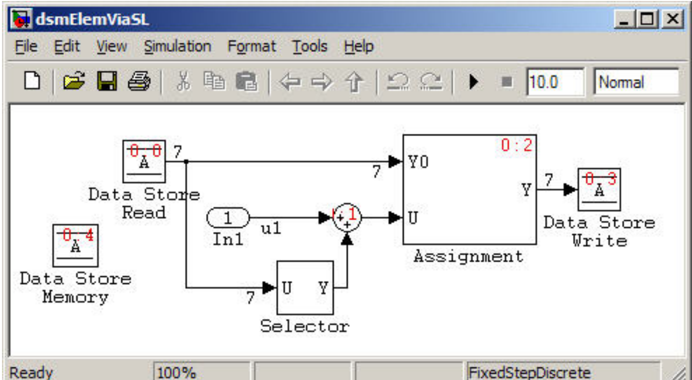
Nonscalar Data Stores in Generated Code

Stateflow generates efficient code for accessing individual elements of nonscalar data stores. For example, the next figure shows a data store named `A` that has seven elements. The Stateflow chart assigns the fourth element of the data store from a value computed from the third element. The generated code does not require access of the other elements of `A`.




```
/* Model step function */
void dsmElemViaSF_step(void)
{
  /* Stateflow: '<Root>/Chart' incorporates:
   *   Inport: '<Root>/In1'
   */
  A[3] = u1 + A[2];
}
```

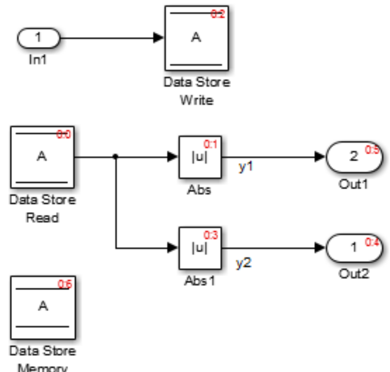
In contrast, modeling and code generation for data store element selection and assignment in Simulink is more explicit. The next figure shows the same algorithm modeled without using a Stateflow chart. The assignment block copies each element of the data store back to itself, in addition to updating the element.



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:

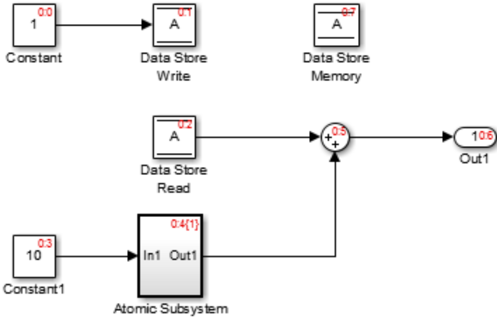


The following execution order applies:

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:



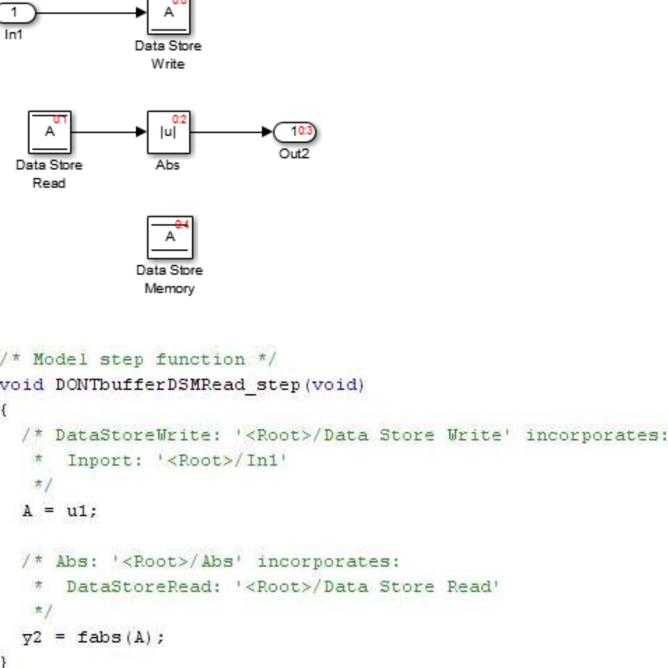
```
real_T rtb_DataStoreRead;

/* 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' */

/* Outport: '<Root>/Out1' incorporates:
 *  Sum: '<Root>/Sum'                Use the buffered value of A
 */
DoBufferDSMRead2_Y.Out1 = rtb_DataStoreRead + DoBufferDSMRead2_B.Abs;
```

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, theSimulink Coder code generator determines that it can optimize away the output buffer for a Data Store Read block, and the generated code will refer 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.

9

Entry-Point Functions and Scheduling

- "Entry-Point Functions and Scheduling" on page 9-2
- "Generate Reentrant Code from Top-Level Models" on page 9-4
- "Generate C++ Class Interface to Model or Subsystem Code" on page 9-5
- "About Model Execution" on page 9-9
- "Non-Real-Time Single-Tasking Systems" on page 9-11
- "Non-Real-Time Multitasking Systems" on page 9-12
- "Real-Time Single-Tasking Systems" on page 9-14
- "Real-Time Multitasking Systems" on page 9-16
- "Multitasking Systems Using Real-Time Tasking Primitives" on page 9-18
- "Program Timing" on page 9-20
- "Program Execution" on page 9-22
- "External Mode Communication" on page 9-23
- "Data Logging in Single-Tasking and Multitasking Model Execution" on page 9-24
- "Rapid Prototyping and Embedded Model Execution Differences" on page 9-25
- "Rapid Prototyping Model Functions" on page 9-26

Entry-Point Functions and Scheduling

The following functions represent entry points in the generated code for a Simulink model.

Function	Description
`model_initialize`	Initialization entry point in generated code for Simulink model. The *model*`_initialize` function performs model initialization and should be called once before you start executing your model.
`model_step`	Step routine entry point in generated code for Simulink model. The *model*`_step` function contains the output and update code for blocks in your model.
`model_terminate`	Termination entry point in generated code for Simulink model. The *model*`_terminate` 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.

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.

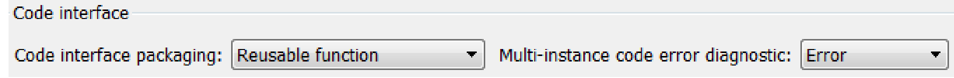
To configure a model to generate reusable, reentrant function code:

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".

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**.

Generate C++ Class Interface to Model or Subsystem Code

In this section...

"About C++ Class Code Interface Packaging" on page 9-5

"Generate C++ Class Interface to Model Code" on page 9-5

"Generate C++ Class Interface to Nonvirtual Subsystem Code" on page 9-7

"C++ Class Interface Limitations" on page 9-7

About C++ Class Code Interface Packaging

Using the **Code interface packaging** option `C++ class`, on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box, you can generate a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods. The benefits of C++ class encapsulation include:

- Greater control over access to model data
- Ability to multiply instantiate model classes
- Easier integration of model code into C++ programming environments

C++ class encapsulation also works for right-click builds of nonvirtual subsystems. (For information on requirements that apply, see "Generate C++ Class Interface to Nonvirtual Subsystem Code" on page 9-7.)

Generate C++ Class Interface to Model Code

To generate encapsulated C++ class code from a GRT-based model:

1. On the **Code Generation** pane of the Configuration Parameters dialog box, set the model parameter **Language** to `C++`. This selection also causes C++ class code interface packaging to be selected for the model.
2. On the **Code Generation** > **Interface** pane, verify that the "**Code interface packaging**" parameter 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_secondOrderSyste_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 methods, see “Entry-Point Functions and Scheduling”.

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”.

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

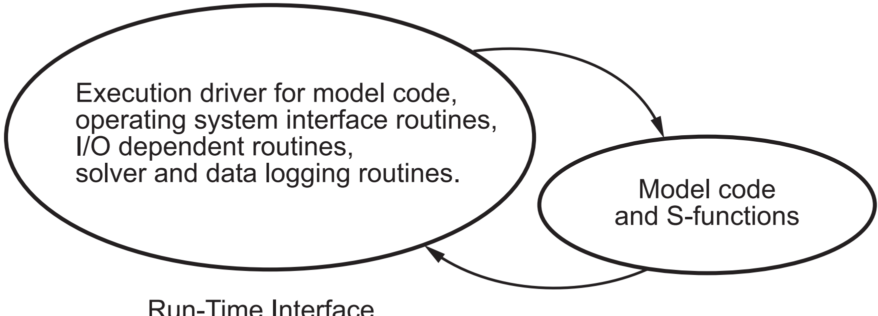
- Among the data exchange interfaces available on the **Interface** pane of the Configuration Parameters dialog box, only the `C API` interface is supported for `C++ class` code generation. If you select `External mode` or `ASAP2`, code generation fails with a validation error.
- If a model root inport 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

About Model Execution

Before looking at the two styles of generated code, you need to have a high-level understanding of how the generated model code is executed. The Simulink Coder software generates algorithmic code as defined by your model. You can include your own code in your model by using S-functions. S-functions can range from high-level signal manipulation algorithms to low-level device drivers.

The Simulink Coder product also provides a run-time interface that executes the generated model code. The run-time interface and model code are compiled together to create the model executable. 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 models, the multitasking environment will provide the most efficient model execution (that is, fastest sample rate).

The following concepts are useful in describing how models execute.

- **Initialization**: `model_initialize` initializes the run-time 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 run-time 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 havs 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.

The identifying names in the preceding list (ModelOutputs, and so on) identify functions in pseudocode examples shown in the following sections.

- "Non-Real-Time Single-Tasking Systems" on page 9-11
- "Non-Real-Time Multitasking Systems" on page 9-12
- "Real-Time Single-Tasking Systems" on page 9-14
- "Real-Time Multitasking Systems" on page 9-16
- "Multitasking Systems Using Real-Time Tasking Primitives" on page 9-18

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 outports.
    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.
    LogTXY                -- 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 outports.
  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 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 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
                            -- outports
    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 outports.
    ModelUpdate(tid=0)     -- major time step.
    Loop:                  -- Integration in minor time step for
                           -- models with continuous states.
      ModelDeriviatives
      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".
```

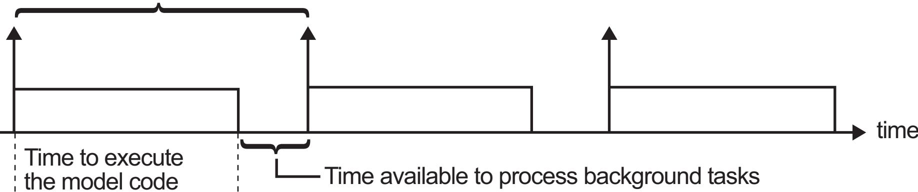
```
  Start clock that does a "semGive" on a clockSem semaphore.
  Wait on "model-running" semaphore.
  Shutdown
}
```

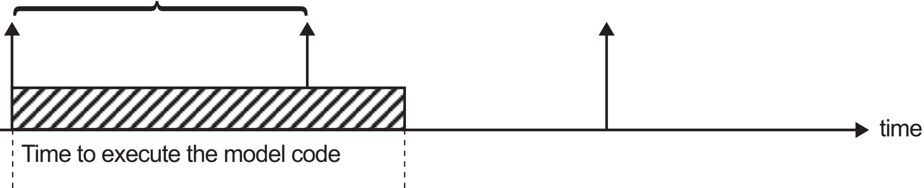
In this multitasking environment, the model is executed using real-time operating system tasking primitives. Such environments require several model tasks (`tBaseRate` and several `tSubRate` tasks) to run the model code. The base rate task (`tBaseRate`) has a higher priority than the subrate tasks. The 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.

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 “Timers” on page 1-69.

Program Execution

As the previous section indicates, 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.

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

The Simulink Coder data-logging features, described in "Debug" on page 10-56, enable you to 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... 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 outport 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.

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 run-time function, `model_step`.

Thus, when you look again at the model execution pseudocode presented earlier in this chapter, you can eliminate the `Loop...EndLoop` statements, and group `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 9-2.

Rapid Prototyping Model Functions

The rapid prototyping code defines the following functions that interface with the run-time interface:

- `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 run-time interface 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 that has a constant sample time.
- `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 run-time interface during major and minor time steps. The major time steps are when the run-time interface 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 run-time 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 run-time 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`.

$$\dot{x} = f_d(t, x_c, u)$$

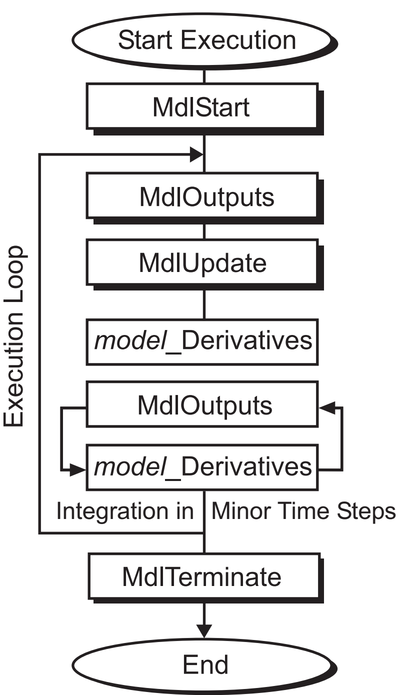
The derivatives x are a function of the current input. Each block that has continuous states provides its derivatives to the solver (for example, `ode5`) in *`model`*`_Derivatives`. The derivatives are used by the solver to integrate the continuous state to produce the next value.

The output, y, is generally written to the block I/O structure. Root-level Outport 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.

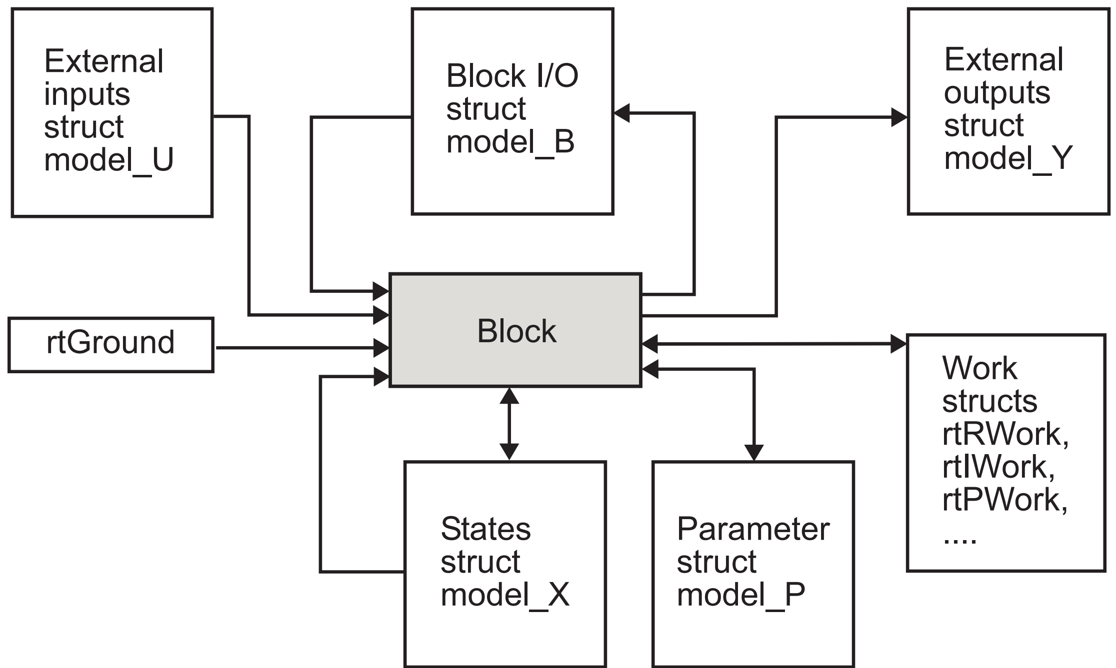
```
/*
 * Version, Model options, TLC options,
 * and code generation information are placed here.
*/
<includes>
void MdlStart(void)
{
  /*
   * State initialization code.
   * Model start-up code - one time initialization code.
   * Execute any block enable methods.
   * Initialize output of any blocks 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 "Signals" on page 8-46 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.

Code Generation

10

Configuration

- “Code Generation Configuration” on page 10-2
- “Open the Model Configuration for Code Generation” on page 10-3
- “Configure a Model Programmatically” on page 10-4
- “Checking Model and Configuration with Model Advisor” on page 10-6
- “Check Model for Code Efficiency” on page 10-7
- “Application Objectives” on page 10-8
- “Target” on page 10-12
- “Change Programming Language” on page 10-46
- “Configure Code Comments” on page 10-47
- “Construction of Generated Identifiers” on page 10-48
- “Identifier Name Collisions and Mangling” on page 10-49
- “Specify Identifier Length to Avoid Naming Collisions” on page 10-50
- “Specify Reserved Names for Generated Identifiers” on page 10-51
- “Reserved Keywords” on page 10-52
- “Debug” on page 10-56

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 Simulink Coder software generates code and builds an executable from your model.

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 "Advice About Optimizing Models for Code Generation" and "Check and Configure Model for Code Generation Objectives Using Configuration Parameters Dialog Box".

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. To open the **Code Generation** pane, see "Open the Model Configuration for Code Generation" on page 10-3. 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 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 "Parameter Command-Line Information Summary" in the Simulink Coder documentation. For an example of automating the configuration of a model, see "Configure a Model Programmatically" on page 10-4.

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 a 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. 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,'InlineParams','on');
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

- “Parameter Command-Line Information Summary”

Checking Model and Configuration with Model Advisor

You can use the Model Advisor checks available with Simulink Coder to assess model readiness for code generation. For information about the Model Advisor, see “Run Model Checks”. For checks available with Simulink Coder, see “Simulink Coder Checks”.

If you want to check and configure your model for code generation objectives such as efficiency or debugging, see “Application Considerations”.

Check Model for Code Efficiency

Before generating code for a model, use the Model Advisor to check the model for conditions and configuration settings that can result in inaccurate or inefficient code.

1. Open `rtwdemo_throttlecntrl` and save a copy as `throttlecntrl` in a writable location on your MATLAB path.
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 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. For checks available with each product, see:

   - "Simulink Coder Checks"
   - "Embedded Coder Checks"
5. In the left pane, select the remaining checks and 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.

Application Objectives

In this section...

"High-Level Code Generation Objectives" on page 10-8
"Check and Configure Model for Code Generation Objectives" on page 10-9
"Check and Configure Model for Code Generation Objectives Using Configuration Parameters Dialog Box" on page 10-11

High-Level Code Generation Objectives

Depending on the type of application that your model represents, you are likely to have specific code generation objectives. For example, execution efficiency might be more critical than debugging. If you have specific objectives, you can quickly configure your model to meet those objectives by selecting and prioritizing from these code generation objectives:

- Execution efficiency (all targets) — Configure code generation settings to achieve fast execution time.
- ROM efficiency (ERT-based targets) — Configure code generation settings to reduce ROM usage.
- RAM efficiency (ERT-based targets) — Configure code generation settings to reduce RAM usage.
- Traceability (ERT-based targets) — Configure code generation settings to provide mapping between model elements and code.
- Safety precaution (ERT-based targets) — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.
- Debugging (all targets) — Configure code generation settings to debug the code generation build process.
- MISRA-C:2004 guidelines (ERT-based targets) — Configure code generation settings to increase compliance with MISRA-C:2004 guidelines.
- Polyspace (ERT-based targets) — Configure code generation settings to prepare the code for Polyspace® analysis.

Based on your objective selections and prioritization, the Code Generation Advisor checks your model and suggests changes that you can make to achieve your code generation objectives.

Note: If you select the MISRA-C:2004 guidelines code generation objective, the Code Generation Advisor checks:

- The model configuration settings for compliance with the MISRA-C:2004 configuration setting recommendations.

- For blocks that are not supported or recommended for MISRA-C:2004 compliant code generation.

Setting code generation objectives and running the Code Generation Advisor provides information on how to meet code generation objectives for your model. The Code Generation Advisor does not alter the generated code. You can use the Code Generation Advisor to make the suggested changes to your model. The generated code is changed only after you modify your model and regenerate code. If you use the Code Generation Advisor to set code generation objectives and check your model, the generated code includes comments identifying which objectives you specified, the checks the Code Generation Advisor ran on the model, and the results of running the checks.

The Code Generation Advisor is not available for a model using a “configuration reference”.

Embedded Coder provides additional capabilities with the Code Generation Advisor, for more information, see “Application Objectives” in the Embedded Coder documentation.

Check and Configure Model for Code Generation Objectives

This example shows how to configure and check your model to meet code generation objectives:

1 On the menu bar, select **Code** > **C/C++ Code** > **Code Generation Advisor**. Alternatively:

   - On the toolbar drop-down list, select `Code Generation Advisor`.
   - Right-click a subsystem, and then select **C/C++ Code** > **Code Generation Advisor**. Go to step 3.

2 In the System Selector window, select the model or subsystem that you want to review, and then click **OK**.

3. In the Code Generation Advisor, on the **Code Generation Objectives** pane, select the code generation objectives 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.

Check and 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.

To run the Code Generation Advisor during code generation, on the **Code Generation** pane, set the **Check model before generating code** parameter to either:

- `On (stop for warnings)` - Code generation stops with a check warning. The Code Generation Advisor dialog box opens as described in step 5.
- `On (proceed with warnings)` - Code generation proceeds with check warnings. The Code Generation Advisor interface opens with a list of the checks it ran on the model, along with the results.

For more information, see "Set Objectives — Code Generation Advisor Dialog Box"

Target

In this section...
“Hardware” on page 10-12
“Available Targets” on page 10-12
“About Targets and Code Formats” on page 10-15
“Types of Target Code Formats” on page 10-16
“Targets and Code Formats” on page 10-27
“Targets and Code Styles” on page 10-28
“Backwards Compatibility of Code Formats” on page 10-29
“Select a Target” on page 10-32
“Template Makefiles and Make Options” on page 10-35
“Custom Targets” on page 10-41
“Standard Math Libraries” on page 10-41
“Change the Standard Math Library” on page 10-41
“Specify Target Interfaces” on page 10-41

Hardware

When you use Simulink software to create and execute a model, and Simulink Coder software to generate code, you may need to 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 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”.

Available Targets

The following table lists supported system target files and their associated code formats. The table also gives references to relevant manuals or chapters in this book.

Note You can select from a range of targets of using the System Target File Browser. This allows you to experiment with configuration options and save your model with different configurations. However, you cannot build or generate code for non-GRT targets unless you have the required license on your system (Embedded Coder license for ERT, Real-Time Windows Target license for RTWIN, and so on).

Selecting a system target file for your model selects either a toolchain or one or more template makefiles as the basis for generating model code.

If the system target file supports toolchain controls, toolchain parameters define the compiler and other tools to be used to generate code.

If the system target file supports template makefile controls, the template makefile that is invoked to generate code activates a compiler.

Targets Available from the System Target File Browser

Target/Code Format	System Target File	Toolchain / Template Makefile and Comments	Reference
Embedded Coder (for PC or UNIX[a] platforms)	`ert.tlc` `ert_shrlib.tlc`	Toolchain specified using **Code Generation** pane or equivalent command-line model configuration parameters.	“Code Generation Targets” (Embedded Coder topic)
Create Visual C++®[b] Solution File for Embedded Coder	`ert.tlc`	`RTW.MSVCBuild` Creates and builds Visual C++ Solution (`.sln`) file	“Code Generation Targets” (Embedded Coder topic)
Embedded Coder for AUTOSAR	`autosar.tlc`	`ert_default_tmf`	“Code Generation Targets” (Embedded Coder topic)
Generic Real-Time Target (for PC or UNIX platforms)	`grt.tlc`	Toolchain specified using **Code Generation** pane or equivalent command-line model configuration parameters.	“Targets and Code Formats” on page 10-27

Target/Code Format	System Target File	Toolchain / Template Makefile and Comments	Reference
Create Visual C++ Solution File for Simulink Coder	`grt.tlc`	`RTW.MSVCBuild` Creates and builds Visual C++ Solution (`.sln`) file	"Targets and Code Formats" on page 10-27
Rapid Simulation Target (default for PC or UNIX platforms)	`rsim.tlc`	`rsim_default_tmf`	"Rapid Simulations" on page 14-2
Rapid Simulation Target for LCC compiler	`rsim.tlc`	`rsim_lcc.tmf`	"Rapid Simulations" on page 14-2
Rapid Simulation Target for UNIX platforms	`rsim.tlc`	`rsim_unix.tmf`	"Rapid Simulations" on page 14-2
Rapid Simulation Target for Visual C++ compiler	`rsim.tlc`	`rsim_vc.tmf`	"Rapid Simulations" on page 14-2
S-Function Target for PC or UNIX platforms	`rtwsfcn.tlc`	`rtwsfcn_default_tmf`	"Generated S-Function Block" on page 14-31
S-Function Target for LCC	`rtwsfcn.tlc`	`rtwsfcn_lcc.tmf`	"Generated S-Function Block" on page 14-31
S-Function Target for UNIX platforms	`rtwsfcn.tlc`	`rtwsfcn_unix.tmf`	"Generated S-Function Block" on page 14-31
S-Function Target for Visual C++ compiler	`rtwsfcn.tlc`	`rtwsfcn_vc.tmf`	"Generated S-Function Block" on page 14-31
ASAM-ASAP2 Data Definition Target	`asap2.tlc`	`asap2_default_tmf`	"ASAP2 Data Measurement and Calibration" on page 17-158
Real-Time Windows Target	`rtwin.tlc` `rtwinert.tlc`	`rtwin.tmf` `rtwinert.tmf`	"Use Simulink Models" (Real-Time Windows Target topic)

Target/Code Format	System Target File	Toolchain / Template Makefile and Comments	Reference

Simulink Real-Time	`slrt.tlc`
	`slrtert.tlc`
		`slrt_default_tmf`
		`slrtert_default_tmf`
		`slrt_vc.tmf`
		`slrtert_vc.tmf`
			“Simulink Real-Time Options Configuration Parameters” (Simulink Real-Time topic)

IDE Link capability	`idelink_grt.tlc` `idelink_ert.tlc`	N/A	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.

Targets Supporting Nonzero Start Time

When you try to build models with a nonzero start time, if the selected target does not support a nonzero start time, the Simulink Coder software does not generate code and displays an error message. The Rapid Simulation (RSim) target supports a nonzero start time when the **RSim Target** > **Solver selection** parameter in the Configuration Parameters dialog box is set to `Use Simulink solver module`. The other listed targets do not support a nonzero start time.

About Targets and Code Formats

A *target* (such as the GRT target) is an environment for generating and building code intended for execution on a certain hardware or operating system platform. A target is defined at the top level by a system target file, which in turn invokes other target-specific files.

A *code format* (such as embedded or real-time) is one property of a target. The code format controls decisions made at several points in the code generation process. These 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. In general, the Embedded-C code format is more efficient than the RealTime code format. Embedded-C code format provides more compact data structures, a simpler calling interface, and static memory allocation. These characteristics make the Embedded-C code format the preferred choice for production code generation.

Before Release 14, only the ERT target and targets derived from the ERT target used the Embedded-C code format. Non-ERT targets used other code formats (for example, RealTime).

Beginning in R14, the GRT target uses the Embedded-C code format for back end code generation. This includes generation of both algorithmic model code and supervisory timing and task scheduling code. Between R14 and R2012a, the GRT target (and derived targets) generated a RealTime code format wrapper around the Embedded-C code. This wrapper provided a calling interface that was backward compatible with existing GRT-based custom targets. The wrapper calls were compatible with the main program module of the GRT target (`grt_main.c` or `grt_main.cpp`). This use of wrapper calls incurred some calling overhead; the pure Embedded-C calling interface generated by the ERT target was more highly optimized.

Beginning in R2012a, GRT targets generate code with the same optimized Embedded-C call interface as ERT targets. This simplifies the task of interacting with the generated code. Target authors can author simpler `main.c` or `.cpp` programs for GRT targets. Also, it is not required to author different main programs for GRT and ERT targets.

For a description of the optimized call interface generated by default for both the GRT and ERT targets, see “Entry-Point Functions and Scheduling”.

Code format unification has simplified the conversion of pre-R2012a GRT-based custom targets to ERT-based targets. See “Make Pre-R2012a Custom GRT-Based Targets ERT-Compatible” on page 10-25 for a discussion of target conversion issues.

Types of Target Code Formats

- “Real-Time Code Format” on page 10-19
- “S-Function Code Format” on page 10-21
- “Embedded Code Format” on page 10-22

Your choice of code format is the most important code generation option. The code format specifies the overall framework of the generated code and determines its style.

When you choose a target, you implicitly choose a code format. Typically, the system target file will specify the code format by assigning the TLC variable `CodeFormat`. The following example is from `ert.tlc`.

```
%assign CodeFormat = "Embedded-C"
```

If the system target file does not assign `CodeFormat`, the default is `RealTime` (as in `grt.tlc`).

If you are developing a custom target, you must consider which code format is best for your application and assign `CodeFormat` accordingly.

Choose the `RealTime` code format for rapid prototyping. If your application does not have significant restrictions in code size, memory usage, or stack usage, you might want to continue using the generic real-time (GRT) target throughout development.

For production deployment, and when your application demands that you limit source code size, memory usage, or maintain a simple call structure, you should use the Embedded-C code format. Consider using the Embedded Coder product, if you need added flexibility to configure and optimize code.

Finally, you should choose the Model Reference or the S-function formats if you are not concerned about RAM and ROM usage and want to

- Use a model as a component, for scalability
- Create a proprietary S-function MEX-file object
- Interface the generated code using the S-function C API
- Speed up your simulation

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, Wind River Systems Tornado, Real-Time Windows Target, Texas Instruments™ DSP, ...

The following table summarizes the various options available for each Simulink Coder code format/target, with the exceptions noted.

Features Supported by Simulink Coder Targets and Code Formats

Feature	GRT	ERT	ERT Shared Library	Wind River Systems VxWorks / Tornado	S- Func	RSIM	RT Win	SLRT	Other Supported Targets[1]
Static memory allocation	X	X		X			X	X	X
Dynamic memory allocation	X[4, 5]			X	X	X		X	
Continuous time	X	X		X	X	X	X	X	
C/C++ MEX S-functions (noninlined)	X	X		X	X	X	X	X	
S-function (inlined)	X	X		X	X	X	X	X	X
Minimize RAM/ROM usage		X		X[2]				X[2]	X
Supports external mode	X	X		X		X	X	X	
Rapid prototyping	X			X			X	X	X
Production code		X		X[2]				X[2]	X[3]
Batch parameter tuning and Monte Carlo methods			X			X			
System-level Simulator			X						

Feature	GRT	ERT	ERT Shared Library	Wind River Systems VxWorks / Tornado	S- Func	RSIM	RT Win	SLRT	Other Supported Targets[1]
Executes in hard real time	X[3]	X[3]		X			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]The Embedded Targets capabilities in Simulink Coder support other targets.

[2]Does not apply to GRT based targets. Applies only to an ERT based target.

[3]The 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.

[4]You 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.

[5]You must select the value **Reusable function** for "**Code interface packaging**" in the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box.

Real-Time Code Format

- "About Real-Time Code Format" on page 10-20
- "Unsupported Blocks" on page 10-21

- “System Target Files” on page 10-21
- “Template Makefiles” on page 10-21

About Real-Time Code Format

The real-time code format (corresponding to the generic real-time target) is useful for rapid prototyping applications. If you want to generate real-time code while iterating model parameters rapidly, you should begin the design process with the generic real-time target. The real-time code format supports:

- Continuous time
- Continuous states
- C/C++ MEX S-functions (inlined and noninlined)

For more information on inlining S-functions, see “Insert S-Function Code” on page 16-40 and “Inlining S-Functions”.

By default, the real-time code format declares memory statically, that is, at compile time. However, if you select the value **Reusable function** for the model configuration parameter “**Code interface packaging**”, the real-time code format supports the following additional capabilities:

- Declare memory dynamically.

  For MathWorks blocks, `malloc` calls are limited to the model initialization code. Generated code is designed to be free from memory leaks, provided that the model termination function is called.
- Deploy multiple instances of the same model with each instance maintaining its own unique data.
- Combine multiple models together in one executable. For example, to integrate two models into one larger executable, the real-time code format maintains a unique instance of each of the two models. If you do not use **Reusable function** code interface packaging, the code generator will not necessarily create uniquely named data structures for each model, potentially resulting in name clashes.

  `rt_malloc_main.c`, the main routine for the generic real-time (GRT) target with **Reusable function** code interface packaging selected, supports one model by default. See “Use GRT with Reusable Function Packaging to Combine Models” for information on modifying `rt_malloc_main.c` to support multiple models. `rt_malloc_main.c` is located in the folder *`matlabroot`*`/rtw/c/src/common`.

Unsupported Blocks

The real-time format does not support the following built-in user-defined blocks:

- Interpreted MATLAB Function block (note that Fcn blocks *are* supported)
- S-Function block — MATLAB language S-functions, Fortran S-functions, or C/C ++ MEX S-functions that call into the MATLAB environment (Fcn block calls *are* supported)

System Target Files

- `grt.tlc - Generic Real-Time Target`
- `rsim.tlc - Rapid Simulation Target`
- `tornado.tlc - Tornado (VxWorks) Real-Time Target`

Template Makefiles

- `grt`
  - `grt_lcc.tmf` — Lcc compiler
  - `grt_unix.tmf` — The Open Group UNIX host
  - `grt_vc.tmf` — Microsoft Visual C++
- `rsim`
  - `rsim_lcc.tmf` — Lcc compiler
  - `rsim_unix.tmf` — UNIX host
  - `rsim_vc.tmf` — Visual C++
- `tornado.tmf`

S-Function Code Format

The S-function code format (corresponding to the S-function target) generates code that conforms to the Simulink MEX S-function API. Using the S-function target, you can build an S-function component and use it as an S-Function block in another model.

The S-function code format is also used by the accelerated simulation target to create the Accelerator MEX-file.

In general, you should not use the S-function code format in a system target file. However, you might need to do special handling in your inlined TLC files to account for

this format. You can check the TLC variable `CodeFormat` to see if the current target is a MEX-file. If `CodeFormat = "S-Function"` and the TLC variable `Accelerator` is set to 1, the target is an accelerated simulation MEX-file.

See "Generated S-Function Block" on page 14-31, for more information.

Embedded Code Format

- "About Embedded Code Format" on page 10-22
- "Use the Real-Time Model Data Structure" on page 10-22
- "Make Pre-R2012a Custom GRT-Based Targets ERT-Compatible" on page 10-25
- "Convert Your Target to Use rtModel" on page 10-26
- "Generate Pre-R2012a GRT Wrapper Code" on page 10-27

About Embedded Code Format

The Embedded-C code format corresponds to the Embedded Coder target (ERT), and targets derived from ERT. This code format includes a number of memory-saving and performance optimizations. See the Embedded Coder "Embedded Coder Product Description" and configuration topics such as "Application Objectives", "Code Generation Targets", and "Code Appearance" for details.

Use the Real-Time Model Data Structure

The Embedded-C format 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 Simulink Coder code generator.

You need to use the set of macros `rtmGetxxx` and `rtmSetxxx` to access the real-time model data structure, which is specific to the Simulink Coder product. 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
`rtmGetdX(rtm)`	Get the derivatives of block continuous states
`rtmGetOffsetTimePtr(RT_MDL rtM)`	Return the pointer to vector that stores sample time offsets of the model associated with rtM
`rtmGetNumSampleTimes(RT_MDL rtM)`	Get the number of sample times that a block has
`rtmGetPerTaskSampleHitsPtr(RT_MDL)`	Return a pointer to `NumSampleTime` × `NumSampleTime` matrix
`rtmGetRTWExtModeInfo(RT_MDL rtM)`	Return an external mode information data structure of the model (used internally for External mode).
`rtmGetRTWLogInfo(RT_MDL)`	Return a data structure used by Simulink Coder logging (internal use only)
`rtmGetRTWRTModelMethodsInfo(RT_MDL)`	Return a data structure of Simulink Coder real-time model methods information (internal use only)
`rtmGetRTWSolverInfo(RT_MDL)`	Return data structure containing solver information of the model (internal use only)
`rtmGetSampleHitPtr(RT_MDL)`	Return a pointer to Sample Hit flag vector
`rtmGetSampleTime(RT_MDL rtM, int TID)`	Get task sample time
`rtmGetSampleTimePtr(RT_MDL rtM)`	Get pointer to a task sample time
`rtmGetSampleTimeTaskIDPtr(RT_MDL rtM)`	Get pointer to a task ID
`rtmGetSimTimeStep(RT_MDL)`	Return simulation step type ID (`MINOR_TIME_STEP`, `MAJOR_TIME_STEP`)
`rtmGetStepSize(RT_MDL)`	Return the fundamental step size of the model
`rtmGetT(RT_MDL,t)`	Get the current simulation time
`rtmSetT(RT_MDL,t)`	Set the time of the next sample hit
`rtmGetTaskTime(RT_MDL,tid)`	Get the current time for the current task
`rtmGetTFinal(RT_MDL)`	Get the simulation stop time

rtm Macro Syntax	Description
`rtmSetTFinal(RT_MDL,finalT)`	Set the simulation stop time
`rtmGetTimingData(RT_MDL)`	Return a data structure used by timing engine of the model (internal use only)
`rtmGetTPtr(RT_MDL)`	Return a pointer to the current time
`rtmGetTStart(RT_MDL)`	Get the simulation start time
`rtmIsContinuousTask(rtm)`	Determine whether a task is continuous
`rtmIsMajorTimeStep(rtm)`	Determine whether the simulation is in a major step
`rtmIsSampleHit(RT_MDL,tid)`	Determine whether the sample time is hit
`rtmGetErrorStatus(rtm)`	Get the current error status
`rtmSetErrorStatus(rtm,val)`	Set the current error status
`rtmGetErrorStatusPointer(rtm)`	Return a pointer to the current error status
`rtmGetStopRequested(rtm)`	Return whether a stop is requested
`rtmGetBlockIO(rtm)`	Get the block I/O data structure
`rtmSetBlockIO(rtm,val)`	Set the block I/O data structure
`rtmGetContStates(rtm)`	Get the continuous states data structure
`rtmSetContStates(rtm,val)`	Set the continuous states data structure
`rtmGetDefaultParam(rtm)`	Get the default parameters data structure
`rtmSetDefaultParam(rtm,val)`	Set the default parameters data structure
`rtmGetPrevZCSigState(rtm)`	Get the previous zero-crossing signal state data structure
`rtmSetPrevZCSigState(rtm,val)`	Set the previous zero-crossing signal state data structure
`rtmGetRootDWork(rtm)`	Get the DWork data structure
`rtmSetRootDWork(rtm,val)`	Set the DWork data structure
`rtmGetU(rtm)`	Get the root inputs data structure (when root inputs are passed as part of the model data structure)

rtm Macro Syntax	Description
`rtmSetU(rtm,val)`	Set the root inputs data structure (when root inputs are passed as part of the model data structure)
`rtmGetY(rtm)`	Get the root outputs data structure (when root outputs are passed as part of the model data structure)
`rtmSetY(rtm,val)`	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”.

Make Pre-R2012a Custom GRT-Based Targets ERT-Compatible

If you developed a GRT-based custom target in a release before R2012a, it is simple to make your target ERT compatible. By doing so, you can take advantage of many efficiencies.

There are several approaches to ERT compatibility:

- If your installation does not include an Embedded Coder license, you can convert a GRT-based target as described in “Convert Your Target to Use rtModel” on page 10-26. This enables your custom target to support current GRT features, including back end Embedded-C code generation.
- If your installation includes an Embedded Coder license, you can do either of the following:
  - Create an ERT-based target, but continue to use your customized version of `grt_main.c` or `grt_main.cpp` module. To do this, you can configure the ERT target to generate a pre-R2012a GRT calling interface, as described in “Generate Pre-R2012a GRT Wrapper Code” on page 10-27. This lets your target support the full ERT feature set, without changing your GRT-based run-time interface.
  - Reimplement your custom target as a completely ERT-based target, including use of an ERT generated main program. This approach lets your target support the full ERT feature set, without the overhead caused by wrapper calls.

Note If you intend to use custom storage classes (CSCs) with a custom target, you must use an ERT-based target. See “Custom Storage Classes” in the Embedded Coder documentation for detailed information on CSCs.

For details on how GRT targets are made call-compatible with previous Simulink Coder product versions, see “Use the Real-Time Model Data Structure” on page 10-22.

Convert Your Target to Use rtModel

The real-time model data structure (`rtModel`) encapsulates model-specific information in a much more compact form than the `SimStruct`. Many ERT-related efficiencies 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)

To take advantage of such efficiencies, you must update your GRT-based target to use the `rtModel` (unless you already did so for Release 13). The conversion requires changes to your system target file, template makefile, and main program module.

The following changes to the system target file and template makefile are required to use `rtModel` instead of `SimStruct`:

- In the system target file, add the following global variable assignment:

  ```
  %assign GenRTModel = TLC_TRUE
  ```

- In the template makefile, define the symbol `USE_RTMODEL`. See one of the GRT template makefiles for an example.

The following changes to your main program module (that is, your customized version of `grt_main.c` or `grt_main.cpp`) are required to use `rtModel` instead of `SimStruct`:

- Include `rtmodel.h` instead of `simstruc.h`.
- Since the `rtModel` data structure has a type that includes the model name, define the following macros at the top of the main program file:

  ```
  #define EXPAND_CONCAT(name1,name2) name1 ## name2
  ```

```
#define CONCAT(name1,name2) EXPAND_CONCAT(name1,name2)
#define RT_MODEL CONCAT(MODEL,_rtModel)
```

- Change the `extern` declaration for the function that creates and initializes the `SimStruct` to

  ```
  extern RT_MODEL *MODEL(void);
  ```

- Change the definitions of `rt_CreateIntegrationData` and `rt_UpdateContinuousStates` to be as shown in the Release 14 version of `grt_main.c`.
- Change function prototypes to have the argument `'RT_MODEL'` instead of the argument `'SimStruct'`.
- The prototypes for the functions `rt_GetNextSampleHit`, `rt_UpdateDiscreteTaskSampleHits`, `rt_UpdateContinuousStates`, `rt_UpdateDiscreteEvents`, `rt_UpdateDiscreteTaskTime`, and `rt_InitTimingEngine` have changed. Change their names to use the prefix `rt_Sim` instead of `rt_` and then change the arguments you pass in to them.

  See the Release 14 version of `grt_main.c` for the list of arguments passed in to each function.
- Modify macros that refer to the `SimStruct` to now refer to the `rtModel`. `SimStruct` macros begin with the prefix `ss`, whereas `rtModel` macros begin with the prefix `rtm`. For example, change `ssGetErrorStatus` to `rtmGetErrorStatus`.

Generate Pre-R2012a GRT Wrapper Code

The Simulink Coder product supports the **"Classic call interface"** model option. When this option is selected, the Simulink Coder product generates model function calls that are compatible with the main program module of the pre-R2012a GRT target (`grt_main.c` or `grt_main.cpp`). These calls act as wrappers that interface to code generated with R2012a or higher.

This option provides a quick way to use code generated with R2012a or higher with a main program module based on pre-R2012a `grt_main.c` or `grt_main.cpp`.

Targets and Code Formats

The Simulink Coder product provides multiple *code formats*. Each code format specifies a framework for code generation suited for specific applications. The code formats and corresponding application areas are:

- **Real-time** — Rapid prototyping
- **S-function** — Creating proprietary S-function MEX-file objects, code reuse, and speeding up your simulation
- **Model reference** — Creating MEX-file objects from entire models that other models can use, sometimes in place of S-functions
- **Embedded C** — Deeply embedded systems

This chapter discusses the relationship of code formats to the available target configurations, and factors you should consider when choosing a code format and target. This chapter also summarizes the real-time, S-function, model referencing, and embedded C/C++ code formats.

Targets and Code Styles

The Simulink Coder software generates 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. This chapter discusses the program architecture, that is, the structure of code generated by the Simulink Coder code generator, associated with these two styles of code. The next table classifies the targets shipped with the product. For related details about code style and target characteristics, see "Types of Target Code Formats" on page 10-16.

Code Styles Listed by Target

Target	Code Style (Using C or C++ Unless Noted)
Embedded Coder embedded real-time (ERT) target	Embedded — Useful as a starting point when using generated C/C++ code in an embedded application (often referred to as a *production code target*).
Simulink Coder Generic real-time (GRT) target	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 target (RSim)	Rapid prototyping — Non-real-time simulation of your model on your workstation. Useful as a high-speed or batch simulation tool.
S-function target	Rapid prototyping — Creates a C MEX S-function for simulation of your model within another Simulink model.

Target	Code Style (Using C or C++ Unless Noted)
Tornado (VxWorks) real-time target[a]	Rapid prototyping — Runs model in real time using the VxWorks real-time operating system tasking primitives. Also useful as a starting point for targeting a real-time operating system.
Real-Time Windows Target	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.

a. Tornado and VxWorks are registered trademarks of Wind River Systems, Inc.

Third-party vendors supply additional targets for the Simulink Coder product. Generally, these can be classified as rapid prototyping targets. For more information about third-party products, see the MathWorks Connections Program Web page: http://www.mathworks.com/products/connections.

This chapter is divided into three sections. The first section discusses model execution, the second section discusses the rapid prototyping style of code, and the third section discusses the embedded style of code.

Backwards Compatibility of Code Formats

Because GRT targets now use Embedded-C code format, existing applications that depend on the RealTime code format's calling interface could have compatibility issues. To address this, a set of macros is generated (in `model.h`) that maps Embedded-C data structures to the identifiers that RealTime code format used. The following, which can be found in `model.h` files created for a GRT target, describes these identifier mappings:

```
/* Backward compatible GRT Identifiers */
#define rtB                                     model_B
#define BlockIO                                 BlockIO_model
#define rtXdot                                  model_Xdot
#define StateDerivatives                        StateDerivatives_model
#define tXdis                                   model_Xdis
#define StateDisabled                           StateDisabled_model
#define rtY                                     model_Y
#define ExternalOutputs                         ExternalOutputs_model
```

```
#define rtP                             model_P
#define Parameters                      Parameters_model
```

Since the GRT target now uses the Embedded-C code format for back end code generation, many Embedded-C optimizations are available to Simulink Coder users. In general, the GRT and ERT targets now have many more common features, but the ERT target offers additional controls for common features. The availability of features is now determined by licensing, rather than being tied to code format. The following table compares features available with a Simulink Coder license with those available under an Embedded Coder license:

Comparison of Features Licensed with the Simulink Coder Product Versus the Embedded Coder Product

Feature	Simulink Coder License	Embedded Coder License
`rtModel` data structure	• Full `rtModel` structure generated • GRT variable declaration: `rtModel_model model_M_;`	• `rtModel` is optimized for the model • Optional suppression of error status field and data logging fields • ERT variable declaration: `RT_MODEL_model model_M_;`
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

Feature	Simulink Coder License	Embedded Coder License
Optimized data initialization	Not available	Options to suppress generation of unnecessary initialization code for 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 “Data Types”, “User-Defined Data Types”, and “Custom Storage Classes”.)
Target-optimized data types header file	Requires full `tmwtypes.h` header file	Generates optimized `rtwtypes.h` 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 `inf`	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

Feature	Simulink Coder License	Embedded Coder License
ANSI[a]-C/C++ code generation	Supported	Supported
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, external mode, ASAP2	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.

Select a Target

The first step to configuring a model for Simulink Coder code generation is to choose and configure a code generation target. When you select a target, other model configuration parameters change automatically to best serve requirements of the target. For example:

- Code interface parameters
- Build process parameters, such as the toolchain or template makefile
- Target hardware parameters, such as word size and byte ordering

Use the **Browse** button on the **Code Generation** pane to open the System Target File Browser and select a system target file. For a complete list of available target configurations, see "Available Targets" on page 10-12.

If you select a target configuration by using the System Target File Browser, your selection appears in the **System target file** field (`target.tlc`).

If you are using a target configuration that does not appear in the System Target File Browser, enter the name of your system target file in the **System target file** field. Click **Apply** or **OK** to configure for that target.

You also can select a target programmatically from MATLAB code, as described in "Select a System Target File Programmatically" on page 10-34.

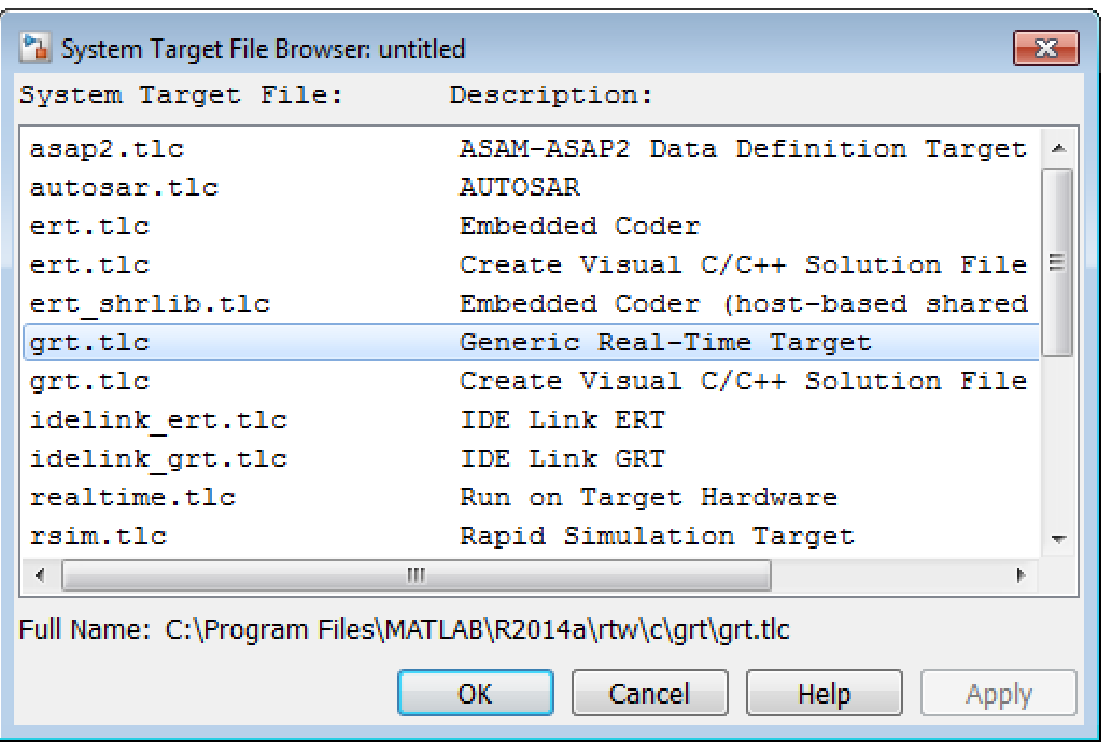
After selecting a target, you can modify model configuration parameter settings.

If you want to switch between different targets in a single workflow for different code generation purposes (for example, rapid prototyping versus product code deployment), set up different configuration sets for the same model and switch the active configuration set for the current operation. For more information on how to set up configuration sets and change the active configuration set, see "Manage a Configuration Set" in the Simulink documentation.

To select a target configuration 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 target configurations, including customizations. When you select a target configuration, the Simulink Coder software 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 target configuration, the Simulink Coder software 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 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 target and the current values of target options. Some of this information is loaded from a system target file when you select a target using the System Target File Browser. You can configure models to generate alternative target code by copying and modifying old or adding new configuration sets and browsing to select a new target.

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 string variables that correspond to the required Simulink Coder system target file, toolchain or template makefile, and/or `make` command settings. For example, for the ERT target, you would define variables for the strings `'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);
```

Template Makefiles and Make Options

The Simulink Coder product includes a set of built-in template makefiles that are designed to build programs for specific targets.

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 target 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. "Compiler or IDE Selection and Configuration" on page 17-2 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.

You can supply options to makefiles 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.

Complete details on the structure of template makefiles are provided in "Customize Template Makefiles" on page 26-56. This section describes compiler-specific template makefiles and common options you can use with each.

Note: To control compiler optimizations for your Simulink Coder makefile build at the Simulink GUI level, use the **Compiler optimization level** option on the **Code Generation** pane of the Configuration Parameters dialog box. The **Compiler optimization level** option 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 Simulink Coder 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®[3] Std 1003.2-1992 (POSIX) standard.

- `ert_unix.tmf`
- `grt_unix.tmf`
- `rsim_unix.tmf`
- `rtwsfcn_unix.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 `-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"
  ```

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 10-38
- “Visual C++ Code Generation Only” on page 10-38

3. IEEE is a registered trademark of The Institute of Electrical and Electronics Engineers, Inc.

Visual C++ Executable Build

To build an executable using the Visual C++ compiler within the Simulink Coder 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"
  ```

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"
  ```

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for the LCC Compiler

The Simulink Coder product 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`.

These options are also documented in the comments at the head of the respective template makefiles.

Enable Build When Path Names Contain Spaces

The Simulink Coder software is able to handle path names that include spaces. Spaces might appear in the path from several sources:

- Your MATLAB installation folder
- The current MATLAB folder in which you initiate a build
- A compiler you are using for a Simulink Coder build

If your work environment includes one or more of the preceding scenarios, use the following support mechanisms as they apply:

- Add the following code to your template makefile (`.tmf`):

  ```
  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 two tokens are not equal, this code replaces `MATLAB_ROOT` with `ALT_MATLAB_ROOT` indicating the path for your MATLAB installation folder includes spaces. Likewise, `ALT_MATLAB_BIN` replaces `MATLAB_BIN`.

  The preceding code is specific to `nmake`. For platform-specific examples, see the supplied Simulink Coder template makefiles .

- When using operating system commands, such as `system` or `dos`, enclose paths that specify executables or command parameters in double quotes (" "). For example:

```
system('dir "D:\Applications\Common Files"')
```

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 26-2. and the topics it references. in panes of the Configuration Parameters dialog box.

Standard Math Libraries

By default, the Simulink code generator produces code that calls the C89/C90 (ANSI C) library for math operations. Depending on your language choice, you have the option of changing the standard math library that the code generator uses. Available libraries include:

Library Name	Language Support	Standard
C89/C90 (ANSI)	C, C++	ANSI C89/C90 (default)
C99 (ISO)	C, C++	ISO/IEC 9899:1990
C++03 (ISO)	C++	ISO/IEC 14882:2003

Change the Standard Math Library

By default, the Simulink code generator uses the ANSI C89/C90 C math library when generating C or C++ code. If your compiler supports newer language standards, you can set the "**Standard math library**" parameter to another supported library. To change the library, change the parameter value on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box.

Specify Target Interfaces

Use the **Interface** pane to control which standard math library you want to use when generating code, whether to include one of three APIs in generated code, and other interface options.

To...	Select or Enter...
Specify the standard math library to use when generating code	Select `C89/C90 (ANSI)`, `C99 (ISO)`, or `C++03 (ISO)` for **Standard math library**. 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 `sin()` whether the input argument is double precision or single precision. However, if you select `C99 (ISO)`, the generated code calls the function `sinf()` 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. For more information, see "Standard math library"
Specify an application-specific library to use when generating code	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`. For more information about code replacement libraries, see and "Code replacement library"
Direct where the Simulink Coder code generator should place fixed-point and other utility code	Select `Auto` or `Shared location` for **Shared code placement**. The shared location directs code for utilities to be placed within the `slprj` folder in your working folder, which is used for building model reference targets. If you select `Auto`, • When the model contains Model blocks, places utility code within the `slprj/`*`target`*`/_sharedutils` folder. • When the model does not contain Model blocks, places utility code in the build folder (generally, in *`model`*`.c` or *`model`*`.cpp`).
Specify a string to be added to the variable names used when logging data to MAT-files, to distinguish logging	Enter a prefix or suffix, such as `rt_` or `_rt`, for **MAT-file variable name modifier**. The Simulink Coder code generator prefixes or appends the string to the variable names for system outputs, states, and simulation time specified in the **Data Import/Export** pane. See "Logging" on page 17-99 for information on MAT-file data logging.

To...	Select or Enter...
data from Simulink Coder and Simulink applications	
Specify an API to be included in generated code	Select `C API`, `External mode`, or `ASAP2` for **Interface**. When you select `C API` or `External mode`, other options appear. `C API` can be used together with either `External mode` or `ASAP2`. For details, see "Generate External Mode and C API Data Interfaces" and "Generate ASAP2 and C API Data Interfaces". However, `External mode` and `ASAP2` are mutually exclusive. For more information on working with these interfaces, see "Data Interchange Using the C API" on page 17-125, "Host/Target Communication" on page 17-50, and "ASAP2 Data Measurement and Calibration" on page 17-158.

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 compiler 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 "Code Generation Pane: Interface". For a summary of option dependencies, see "Interface Dependencies" on page 10-43.

Interface Dependencies

Several parameters available on the Interface pane 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.

Parameter	Dependencies?	Dependency Details
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)	No	
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`
Pass root-level I/O as (ERT targets only)	Yes	Set **Code interface packaging** to `Reusable function`

Parameter	Dependencies?	Dependency Details
Use dynamic memory allocation for model initialization (ERT targets only)	Yes	Set **Code interface packaging** to `Reusable function`
MAT-file logging	Yes	For GRT targets, requires that you enable **Support non-finite numbers**; for ERT targets, 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**
Suppress error status in real-time model data structure (ERT targets only)	No	
Interface	No	
Generate C API for: signals	Yes	Set **Interface** to `C API`
Generate C API for: parameters	Yes	Set **Interface** to `C API`
Generate C API for: states	Yes	Set **Interface** to `C API`
Transport layer	Yes	Set **Interface** to `External mode`
MEX-file arguments	Yes	Set **Interface** to `External mode`
Static memory allocation	Yes	Set **Interface** to `External mode`
Static memory buffer size	Yes	Enable **Static memory allocation**

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 “Compiler or IDE Selection and Configuration”.
3. Check whether the standard math library is configured for your compiler. By default, the code generator uses the C89/C90 (ANSI) library. For C, you have the option of using the C99 (ISO) library. For C++, you can use the C99 (ISO) or C++03 (ISO) library.

   For more information, see “Standard math library”.

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	**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 `model_prm.h`. 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.

Note Comments might include international (non-US-ASCII) characters encountered during code generation when found in:

- Simulink block names and block descriptions
- User comments on Stateflow diagrams
- Stateflow object descriptions
- Custom TLC files
- Code generation template files

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
- A unique *name mangling* string

The code generator conditionally generates the name mangling string 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 10-50
- "Specify Reserved Names for Generated Identifiers" on page 10-51

Simulink Coder software reserves certain words for its own use as keywords of the generated code language. For more information, see "Reserved Keywords" on page 10-52.

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* strings are generated and inserted into each of the potentially conflicting identifiers. Each name mangling string 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 string. 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 Model References”.

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, a name mangling string is generated. The string 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 10-52
"C++ Reserved Keywords" on page 10-53
"Reserved Keywords for Code Generation" on page 10-53
"Simulink Coder Code Replacement Library Keywords" on page 10-54

Simulink Coder keywords are reserved for use internal to Simulink Coder software and should not be used in Simulink models as identifiers or function names. C reserved keywords should also not be used in Simulink models as identifiers or function names. If your model contains reserved keywords, the code generation build does not complete and an error message is displayed. To address this error, modify your model to use identifiers or names that are not reserved.

If you are generating C++ code using the Simulink Coder software, your model must not contain both the "Reserved Keywords for Code Generation" on page 10-53 and the "C ++ Reserved Keywords" on page 10-53.

Note: You can register additional reserved identifiers in the Simulink environment. For more information, see "Specify Reserved Names for Generated Identifiers" on page 10-51.

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	fortran	localZCE	rtNaN
asm	HAVESTDIO	localZCSV	SeedFileBuffer
bool	id_t	matrix	SeedFileBufferLen
boolean_T	int_T	MODEL	single
byte_T	int8_T	MT	TID01EQ
char_T	int16_T	NCSTATES	time_T
cint8_T	int32_T	NULL	true
cint16_T	int64_T	NUMST	TRUE
cint32_T	INTEGER_CODE	pointer_T	uint_T
creal_T	LINK_DATA_BUFFER_S	PROFILING_ENABLED	uint8_T
creal32_T	LINK_DATA_STREAM	PROFILING_NUM_SAMP	uint16_T
creal64_T	localB	real_T	uint32_T
cuint8_T	localC	real32_T	uint64_T
cuint16_T	localDWork	real64_T	UNUSED_PARAMETER
cuint32_T	localP	RT	USE_RTMODEL
ERT	localX	RT_MALLOC	VCAST_FLUSH_DATA
false	localXdis	rtInf	vector
FALSE	localXdot	rtMinusInf	

Simulink Coder 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 have 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.getTflReservedIdentifiers()
```

This function returns an array of library keyword strings. Specifying the input argument is optional.

Note: To list the libraries currently registered in your environment, use the MATLAB command `RTW.viewTfl`.

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.getTflReservedIdentifiers('GNU C99 Extensions')
```

Here is a partial example of the function output:

```
>> lib_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('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 keyword strings 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

Use parameters on the **Diagnostics** and **Code Generation** > **Debug** panes of the Configuration Parameter dialog box to configure a model such that generated code and the build process are optimized 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 *model*`.rtw` 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 *model*`.rtw` 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 `-dc` for the **System target file** parameter on the **Code Generation** pane. To start the debugger and run a debugger script, enter `-df filename` for **System target file**.
Generate a report containing statistics indicating how many times the Simulink Coder code generator reads each line of	**Start TLC coverage when generating code**. Alternatively, enter the argument `-dg` for the **System Target File** parameter on the **Code Generation** pane.

To...	Select...
TLC code during code generation	

Halt a build if a user-supplied TLC file contains an `%assert` directive that evaluates to `FALSE`	**Enable TLC assertion**. Alternatively, you can use MATLAB commands to control TLC assertion handling.
	
	To set the flag on or off, use the `set_param` command. The default is off.
	
	`set_param(model, 'TLCAssertion', 'on|off')`
	
	To check the current setting, use `get_param`.
	
	`get_param(model, 'TLCAssertion')`

Detect loss of tunability	**Diagnostics** > **Data Validity** > **Detect loss of tunability**. 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: • `none` — Loss of tunability can occur without notification. • `warning` — Loss of tunability generates a warning (default). • `error` — Loss of tunability generates an error. For a list of supported operators and functions, see “Tunable Expression Limitations”

To...	Select...

Enable model verification (assertion) blocks	**Diagnostics** > **Data Validity** > **Model Verification block enabling** . 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:
	• `User local settings`
	• `Enable All`
	• `Disable All`
	
	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).
	
	`utAssert(input_signal);`
	`utAssert(input_signal != 0.0);`
	`utAssert(input_signal != 0);`
	
	By default, `utAssert` does not change generated code. For assertions to abort execution, you must enable them by specifying the following `make_rtw` command for **Code Generation** > **Make command**:
	
	`make_rtw OPTS="-DDOASSERTS"`
	
	Use the following variant if you want triggered assertions to print the assertion statement instead of aborting execution:
	
	`make_rtw OPTS="-DDOASSERTS -DPRINT_ASSERTS"`
	
	utAssert is defined as `#define utAssert(exp) assert(exp)`.
	
	To customize assertion behavior, provide your own definition of `utAssert` in a handwritten header file that overrides the default `utAssert.h`. For details on how to include a

To...	Select...
	customized header file in generated code, see “Configure Model for External Code Integration”. 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”. Also, consider using the Model Advisor as a tool for troubleshooting model builds.

For descriptions of Debug pane parameters, see “Code Generation Pane: Debug” in the Simulink Coder reference documentation.

Source Code Generation

- “Initiate Code Generation” on page 11-2
- “Model and Test Environment” on page 11-3
- “Configure Model and Generate Code” on page 11-14
- “Configure Data Interface” on page 11-20
- “Call External C Functions” on page 11-29
- “Reload Generated Code” on page 11-36
- “Generated Source Files and File Dependencies” on page 11-37
- “Files and Folders Created by Build Process” on page 11-56
- “How Code Is Generated From a Model” on page 11-63
- “Code Generation of Matrices and Arrays” on page 11-65
- “Generated Code Considerations” on page 11-69
- “About Shared Utility Code” on page 11-71
- “Controlling Shared Utility Code Placement” on page 11-72
- “rtwtypes.h and Shared Utility Code” on page 11-73
- “Incremental Shared Utility Code Generation and Compilation” on page 11-74
- “Shared Utility Checksum” on page 11-75
- “Shared Fixed-Point Utility Functions” on page 11-77
- “Share User-Defined Data Types Across Models” on page 11-79
- “Generating Code Using Simulink® Coder™” on page 11-84

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 click **Generate Code**. To initiate a full model build, clear the **Generate code only** option and click the **Build** button, or use a method described in "Initiate the Build Process" on page 17-18.

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 11-3
"Functional Design of the Model" on page 11-4
"View the Top Model" on page 11-4
"View the Subsystems" on page 11-5
"Simulation Test Environment" on page 11-6
"Run Simulation Tests" on page 11-11
"Key Points" on page 11-12
"Learn More" on page 11-13

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

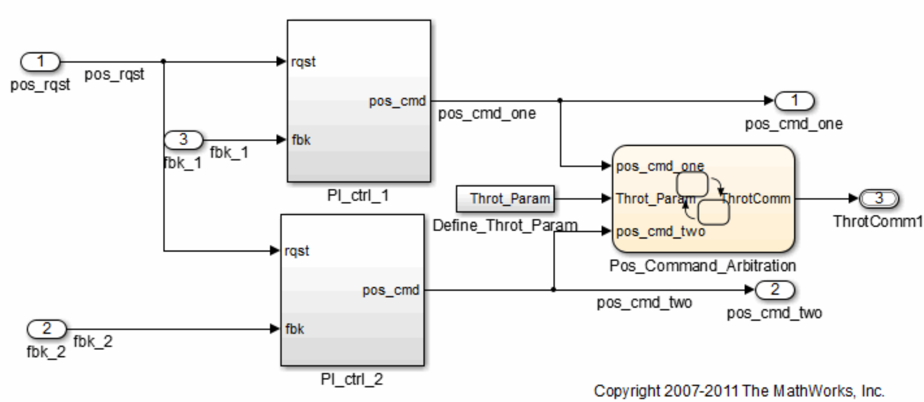
Functional Design of the Model

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	
Omit 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.

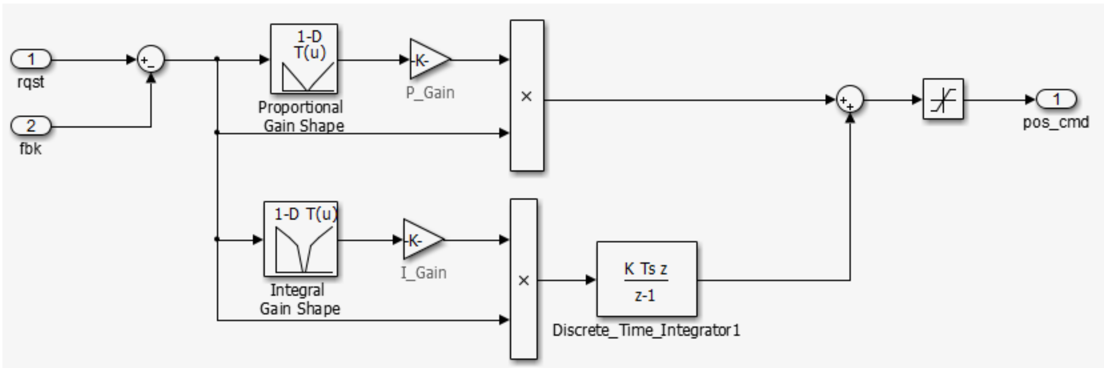
View the Subsystems

Explore two of the subsystems in the top model.

1 If not already open, open `throttlecntrl`.

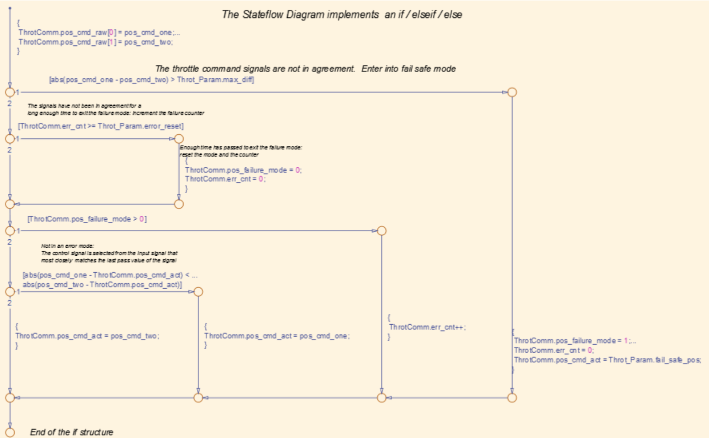
   Two subsystems in the top model represent proportional-integral (PI) controllers, `PI_ctrl_1` and `PI_ctrl_2`. At this stage, these identical subsystems, use identical data. If you have an Embedded Coder license, you can use these subsystems in a example that shows “how to create reusable functions”.

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 11-6. You cannot edit a block that you add to a model from a library. You must 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 `throttlecntrl`.

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:

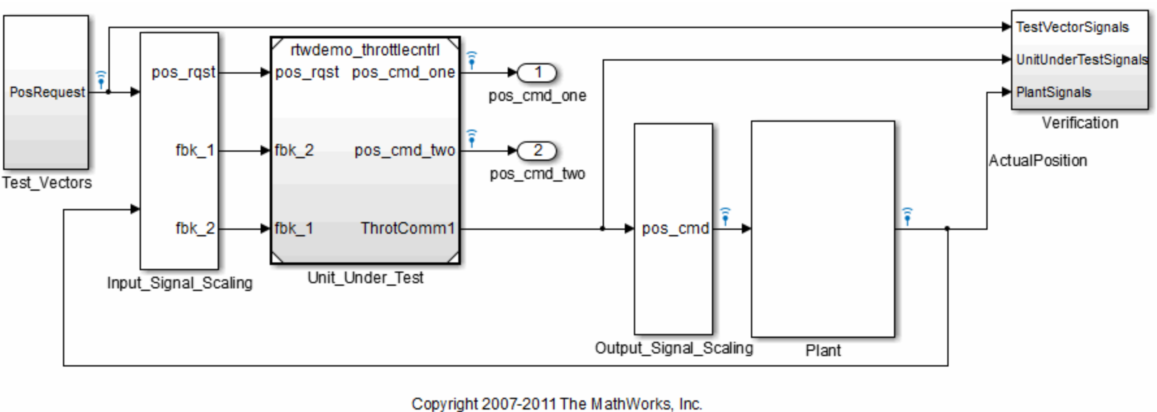
- Separates test data from the control algorithm.
- Separates the plant or feedback model from the control algorithm.
- Provides a reusable environment for multiple versions of the control algorithm.

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.

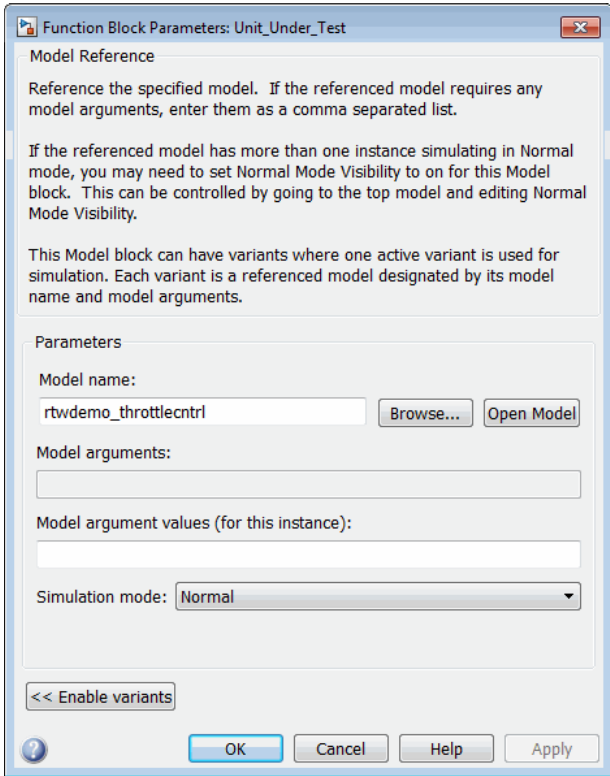
1 Open the test harness model `rtwdemo_throttlecntrl_testharness` and save a copy as `throttlecntrl_testharness` in a writable location on your MATLAB path.



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`.

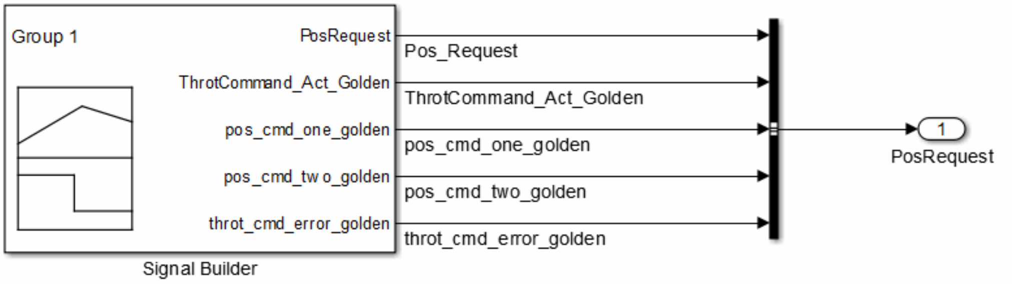
d Update the test harness model diagram by clicking **Simulation** > **Update Diagram**.

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:

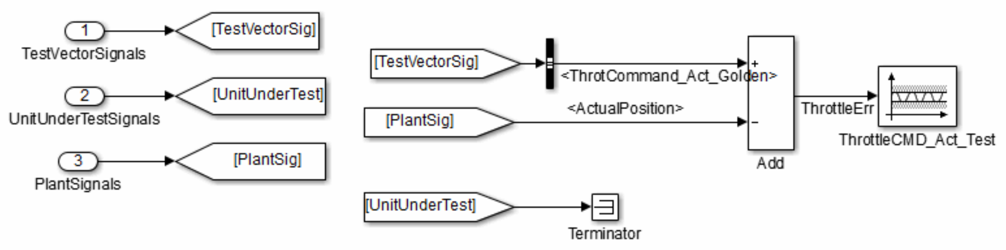
- Simulation time is faster for large models.
- You can directly simulate compiled functions.
- Simulation requires less memory. Only one copy of the compiled model is in memory, even when the model is referenced multiple times.

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`.

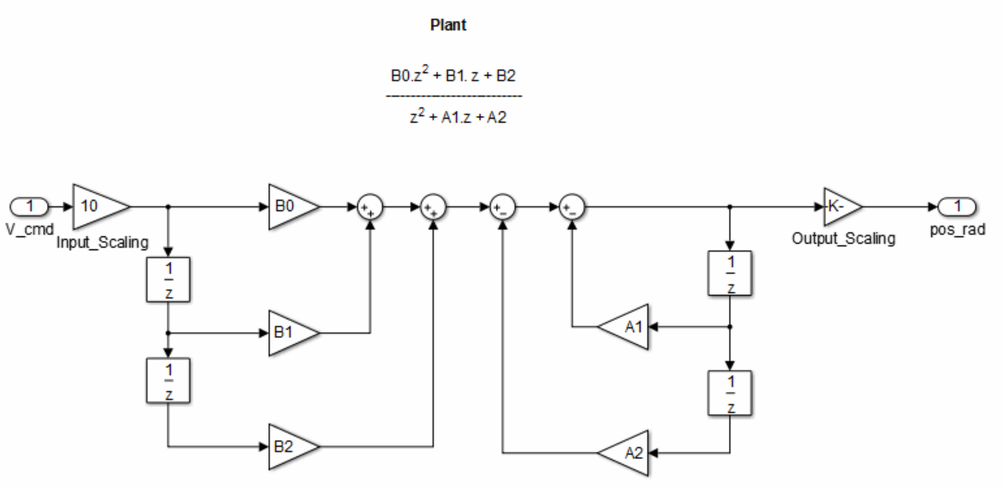


A test harness compares control algorithm simulation results against *golden data* — test results that exhibit the desired behavior for the control algorithm as certified by an expert. In the `Verification` subsystem, an Assertion block compares the simulated throttle value position from the plant against 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.

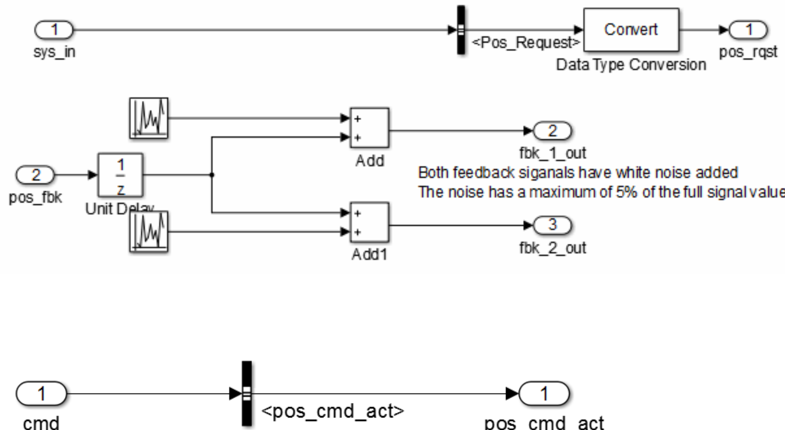
Alternatively, you can evaluate the simulation data after the simulation completes execution. You can use either MATLAB scripts or third-party tools to perform the evaluation. Post-execution evaluation provides greater flexibility in the analysis of data. However, it requires waiting until execution 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 and 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.

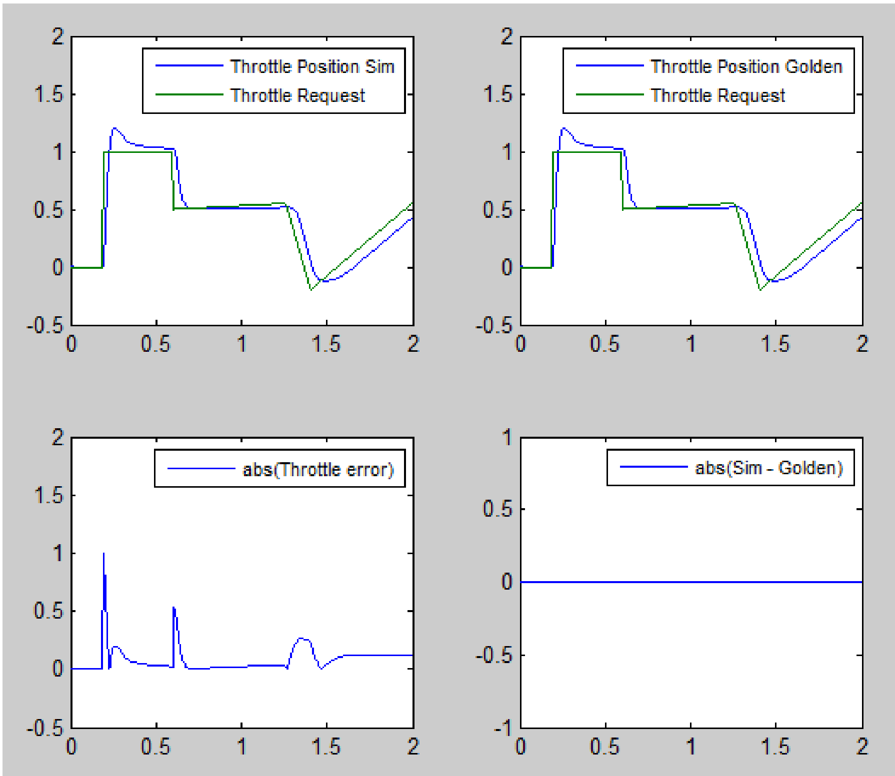
7 Save and close `throttlecntrl_testharness`.

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 should 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, be sure that you 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 and typically 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”
- “Code Generation”
- “Signal Groups”

Configure Model and Generate Code

In this section...

"About This Example" on page 11-14
"Configure the Model for Code Generation" on page 11-2
"Save Your Model Configuration as a MATLAB Function" on page 11-16
"Check the Model for Adverse Conditions and Code Generation Settings" on page 11-17
"Generate Code for the Model" on page 11-17
"Review the Generated Code" on page 11-17
"Generate an Executable" on page 11-18
"Key Points" on page 11-19
"Learn More" on page 11-19

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

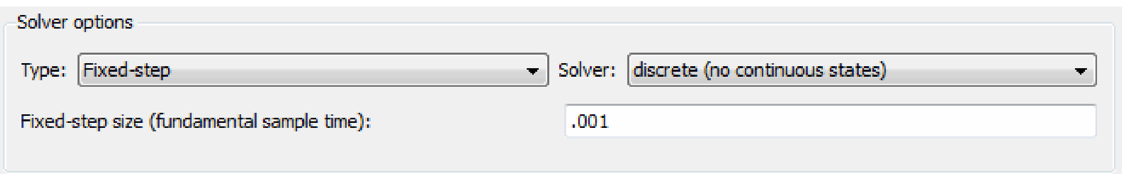
`rtwdemo_throttlecntrl` model file

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 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**. Note the following path appears in the text field:

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

This directory 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 the Model for Adverse Conditions and Code Generation Settings

Before generating code for a model, use the Simulink Model Advisor to check the model for conditions and configuration settings 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 Return to the **Code Generation** pane, click **Generate Code**, 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
`throttlecntrl.c`	C file that contains the scheduler, controller, initialization, and interface code
`throttlecntrl_data.c`	C file that assigns values to generated data structures
`throttlecntrl.h`	Header file that defines data structures
`throttlecntrl_private.h`	Header file that defines data used only by the generated code
`throttlecntrl_types.h`	Header file that defines the model data structure

For more information, see “Generated Source Files and File Dependencies”.

At this point you might also want to consider logging data to a MAT-file. For an example, see “Log Data for Analysis”.

Generate an Executable

1 Open `throttlecntrl`.

2. In the Configuration Parameters dialog box, clear the **Code Generation** > **Generate code only** check box and click **Apply**.
3. Click **Build**. 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 file`throttlecntrl.exe`.
5. Run the executable. In the Command Window, enter `!throttlecntrl`. 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 you might also want to consider logging data to a MAT-file. For an example, see "Log Data for Analysis".

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 then a variable-step solver can be used 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.

Learn More

- "Code Generation"
- "Configuration Reuse"
- "Run Model Checks"

Configure Data Interface

In this section...

"About This Example" on page 11-20

"Declare Data" on page 11-20

"Use Data Objects" on page 11-21

"Add New Data Objects" on page 11-25

"Enable Data Objects for Generated Code" on page 11-25

"Effects of Simulation on Data Typing" on page 11-26

"Manage Data" on page 11-27

"Key Points" on page 11-28

"Learn More" on page 11-28

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

- "Tunable Parameter Storage Classes"
- "Signals Storage Classes"
- "State Storage Classes"

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 a variety of ways. This example focuses on the following types of data objects:

- Signal
- Parameter
- Bus

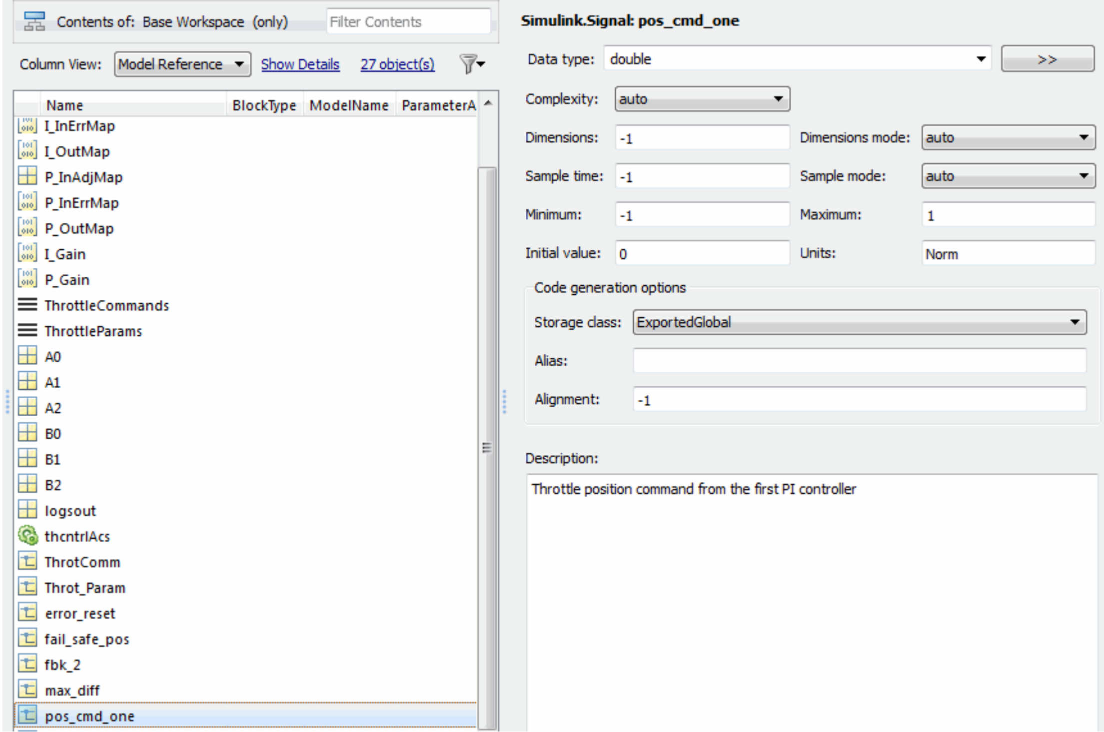
To configure the data interface for your model using the data object method, in the MATLAB base workspace, you define data objects and then associate them with your Simulink model or embedded Stateflow chart. When you build your model, the build process uses the associated base workspace data objects in the generated code.

A data object has a mixture of *active* and *descriptive* fields. Active fields potentially affect simulation or code generation. Descriptive fields do not affect simulation or code generation. They are used with data dictionaries and model-checking tools.

- Active fields:
  - Data type
  - Storage class
  - Value (parameters)
  - Initial value (signals)
  - Alias (define a different name in the generated code)
  - Dimension (inherited for parameters)
  - Complexity (inherited for parameters)
- Descriptive fields:
  - Minimum
  - Maximum
  - Units
  - Description

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 =

Simulink.Signal handle
Package: Simulink

Properties:
       CoderInfo: [1x1 Simulink.SignalCoderInfo]
     Description: 'Throttle position command from the first PI controller'
        DataType: 'double'
```

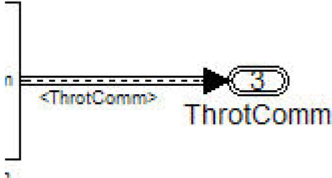
```
            Min: -1
            Max: 1
       DocUnits: 'Norm'
     Dimensions: -1
 DimensionsMode: 'auto'
     Complexity: 'auto'
     SampleTime: -1
   SamplingMode: 'auto'
   InitialValue: '0'

Methods, Events, Superclasses
```

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 below.



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:

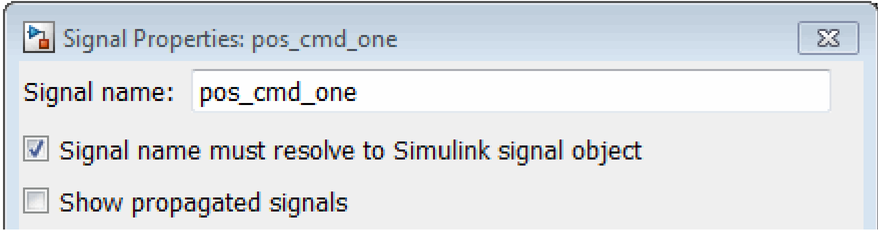
1. In the model window, select **Code** > **Data Objects** > **Data Object Wizard**. The Data Object Wizard dialog box opens.
2. To find candidate constructs, click **Find**. Constructs `fbk_1` and `pos_cmd_two` appear in the dialog box.
3. To select both constructs, click **Check All**.
4. To apply the default Simulink package for the data objects, click **Apply Package**.
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. In the Model Explorer **Model Hierarchy**, expand the `throttlecntrl_datainterface` model node.
2. Click the **Configuration (Active)** node. Make sure that you select, **Optimization** > **Signals and Parameters** > **Inline parameters**.
3. 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.



4. Enable signal object resolution for all signals in the model. In the MATLAB Command Window, enter:

   ```
   disableimplicitsignalresolution('throttlecntrl_datainterface')
   ```

5. Save and close `throttlecntrl_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 by running the test harness.

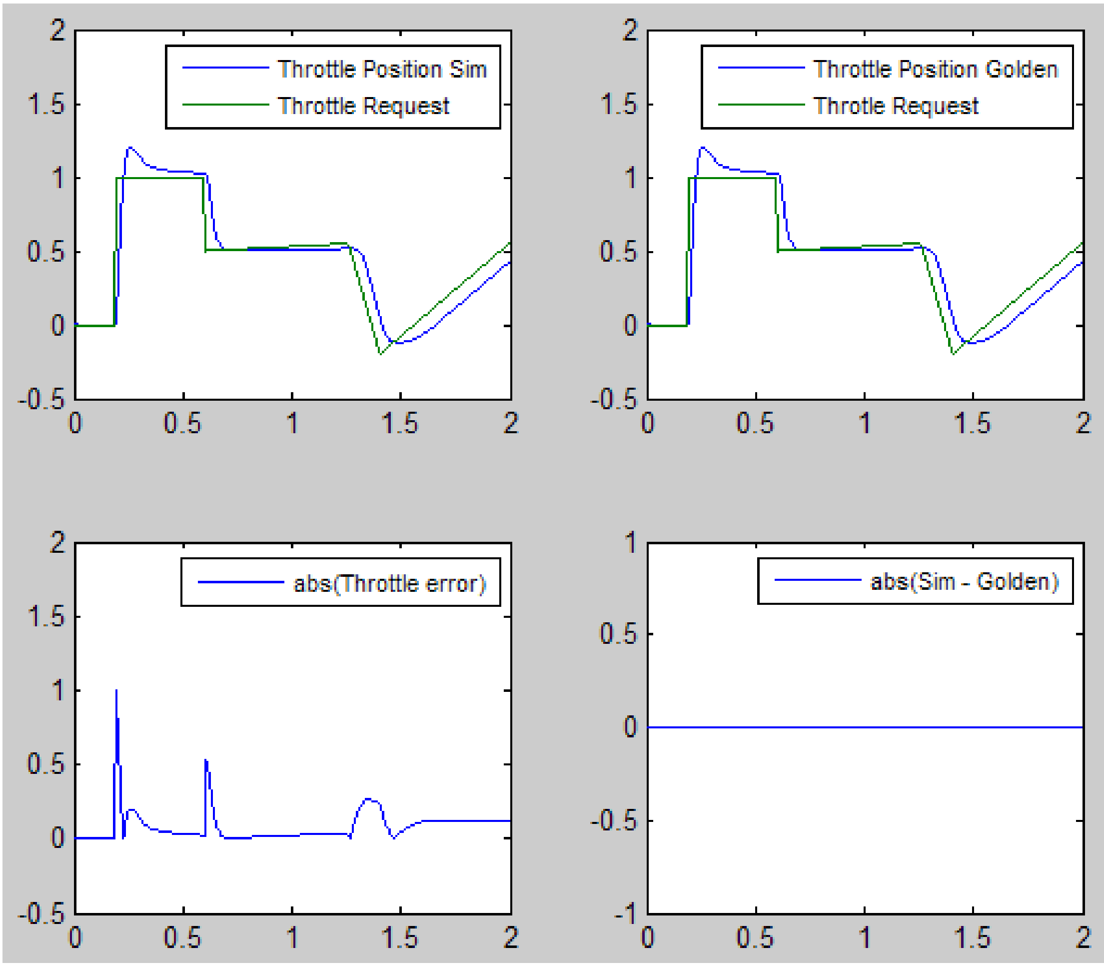
Before you run your test harness, update it to include the `throttlecntrl_datainterface` model.

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 might 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.
- You must explicitly configure data objects to appear by name in generated code.
- Separation of data from model provides several benefits.

Learn More

- “Import Data”
- “Data Representation”
- “Custom Storage Classes”
- “Manage Placement of Data Definitions and Declarations”

Call External C Functions

In this section...

"About This Example" on page 11-29
"Include External C Functions in a Model" on page 11-30
"Create a Block That Calls a C Function" on page 11-30
"Validate External Code in the Simulink Environment" on page 11-32
"Validate C Code as Part of a Model" on page 11-33
"Call a C Function from Generated Code" on page 11-35
"Key Points" on page 11-35
"Learn More" on page 11-35

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 preexisting 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 *`matlabroot`*`/toolbox/rtw/rtwdemos/EmbeddedCoderOverview/stage_4_files`. Put the copies in your working folder.

Note: *`matlabroot`* represents the name of your top-level MATLAB installation 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_simptablewrap`.
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.

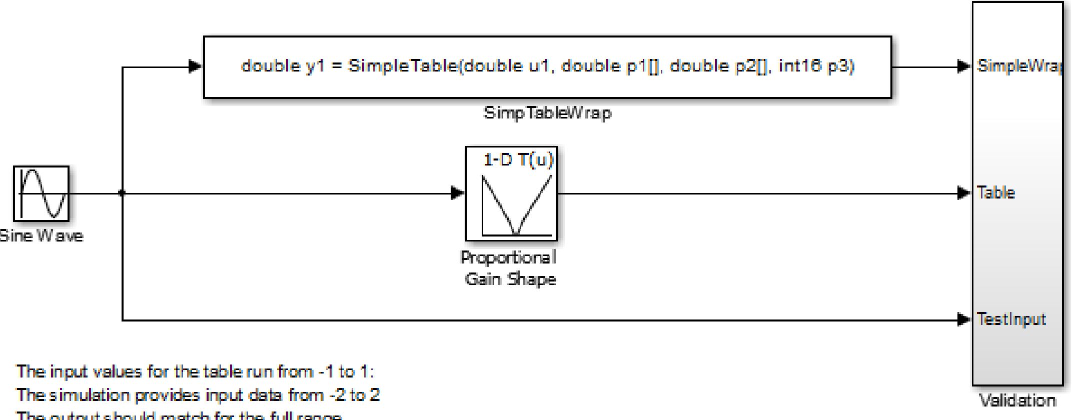
For more information on using the Legacy Code Tool, see:

- "Integrate C Functions Using Legacy Code Tool" in the Simulink documentation
- "Integrate External Code Using Legacy Code Tool"

Validate External Code in the Simulink Environment

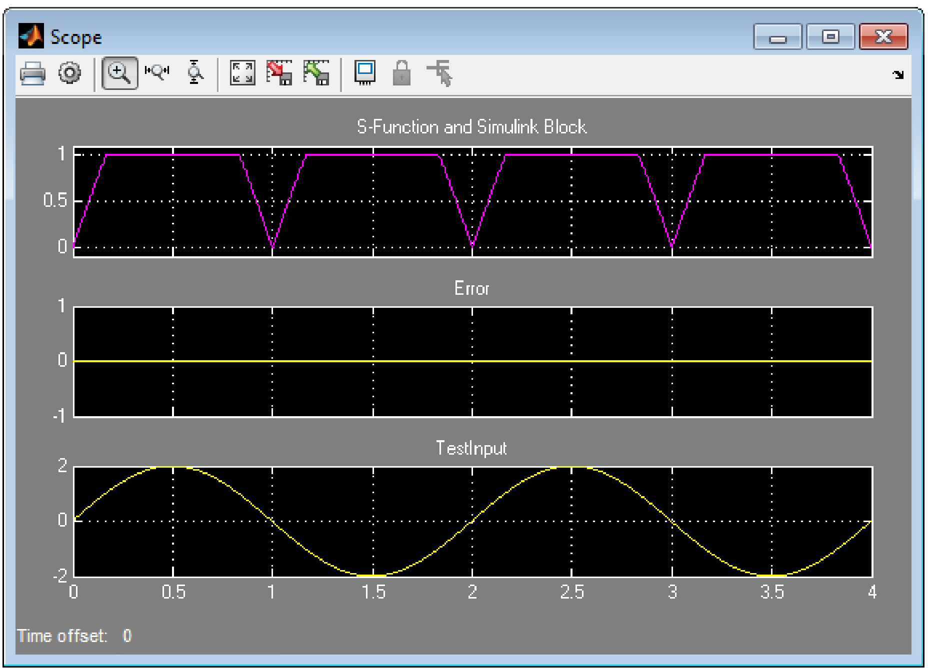
When you integrate external C code with a Simulink model, before using the code, validate the functionality of the external C function code as a standalone component.

1. Open the model `rtwdemo_ValidateLegacyCodeVrsSim`. This model validates the S-function block that you just created.



   - 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_throttlecntrl_extfunccall` and save a copy as `throttlecntrl_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.

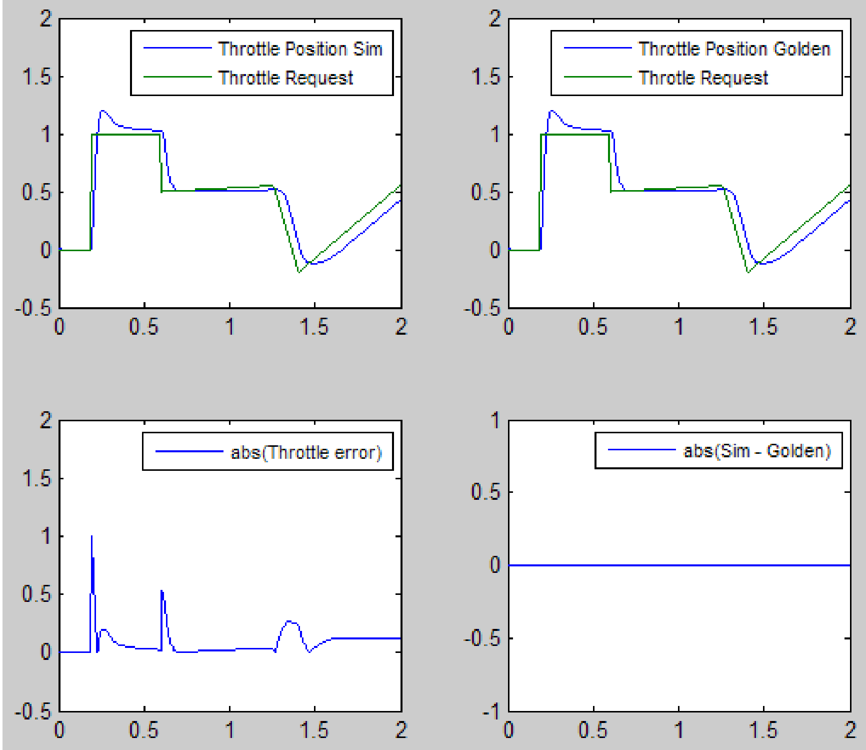
3 Open the test harness model, right-click the `Unit_Under_Test` Model block, and select **Block Parameters (ModelReference)**.

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.

Learn More

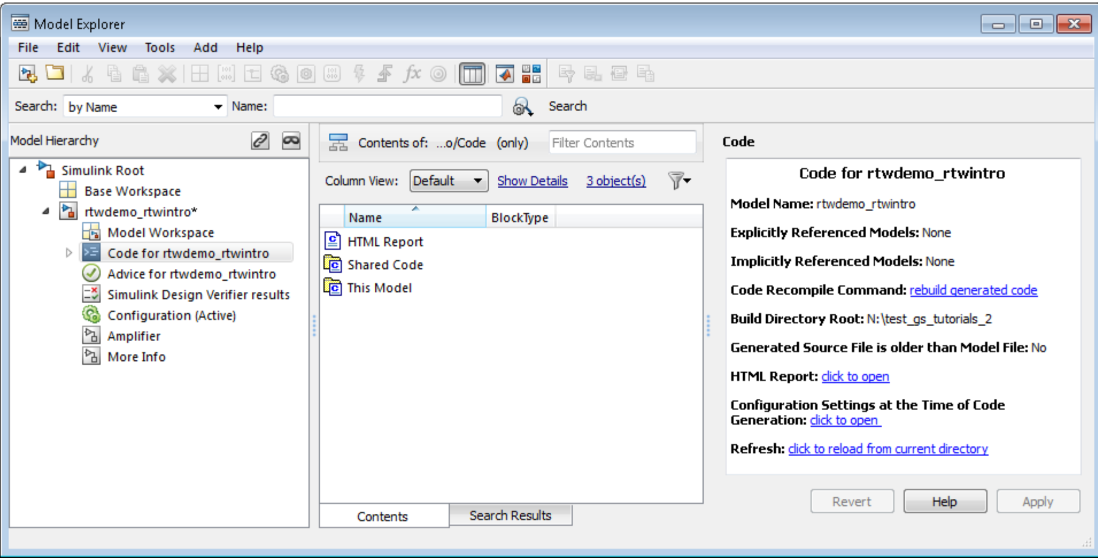
- “Integrate C Functions Using Legacy Code Tool”
- “Insert S-Function Code”

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 Simulink Coder software reloads the code for the model from the build folder.

Generated Source Files and File Dependencies

In this section...

"About Generated Files and File Dependencies" on page 11-37

"Header Dependencies When Interfacing Legacy/Custom Code with Generated Code" on page 11-39

"Dependencies of the Generated Code" on page 11-48

"Specify Include Paths in Simulink Coder Generated Source Files" on page 11-53

About Generated Files and File Dependencies

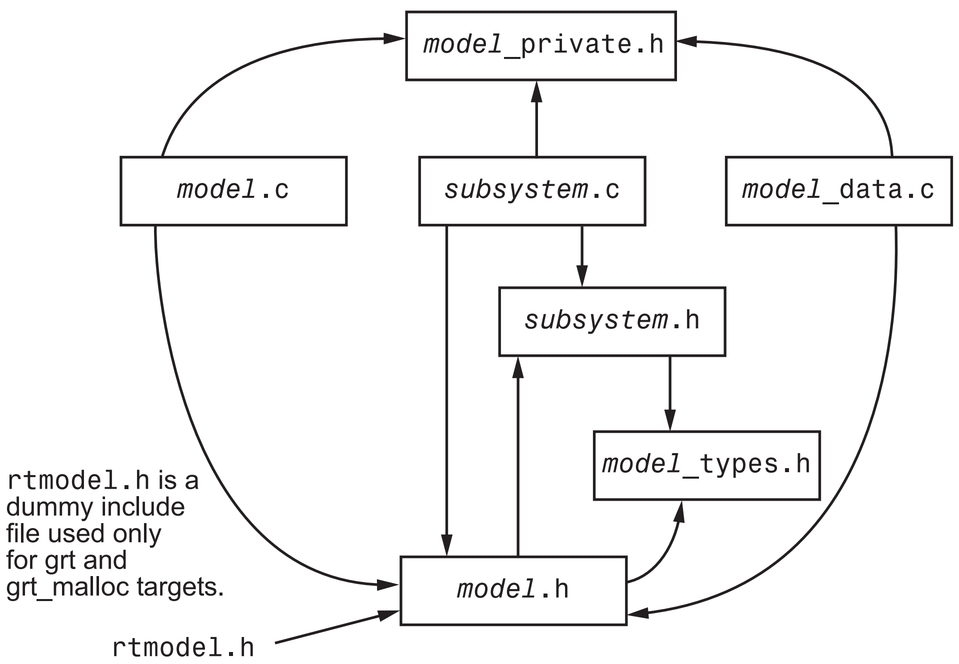
The Simulink Coder software generates code into a set of source files that vary little among different targets. Not all possible files are generated for every model. Some files are created only when the model includes subsystems, calls external interfaces, or uses particular types of data. The Simulink Coder code generator handles most of the code formatting decisions (such as identifier construction and code packaging) in consistent ways.

The source and make files created during the Simulink Coder 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 depend on target settings and options (for example, support files for C API or external mode). See "Files and Folders Created by Build Process" for descriptions of the generated files.

Note The file packaging of Embedded Coder targets differs slightly from the file packaging described below. 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. Utility and model reference code located in a code generation folder might also be referenced by these files. See "Code Generation Folder Structure for Model Reference Targets" on page 4-15 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 descendants' subsystems, as well as to see into the root system (because every *subsystem*.c or *subsystem*.cpp includes *model*.h and *model*_private.h).



Simulink Coder Generated File Dependencies

Note: In the preceding figure, files *model*.h, *model*_private.h, and *subsystem*.h also depend on the Simulink Coder header file rtwtypes.h. Targets that are not based on the ERT target can have additional dependencies on tmwtypes.h and simstruc_types.h.

Header Dependencies When Interfacing Legacy/Custom Code with Generated Code

You can incorporate legacy or custom code into a Simulink Coder build in several ways. One common approach is by creating S-functions. For details on this approach, see "Insert S-Function Code" on page 16-40.

Another approach is to interface code using global variables created by declaring storage classes for signals and parameters. This requires customizing an outer code harness, typically referred to as a `main.c` or `main.cpp` file, to properly execute to the generated code. In addition, the harness can contain custom code.

These scenarios require you to include header files specific to the Simulink Coder product to 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...	rtwtypes.h
GRT target	Provides a complete set of definitions by including `tmwtypes.h` and `simstruc_types.h`, both of which depend on • System headers `limits.h` and `float.h` • Simulink Coder headers: `rtw_matlogging.h`, `rtw_extmode.h`, `rtw_continuous.h`, and `rtw_solver.h`
ERT target and targets based on the ERT target	Is optimized, when possible, to include a minimum set of `#define` statements, enumerations, and type definitions; does not include `tmwtypes.h` and `simstruc_types.h`

The Simulink Coder build process generates the optimized version of `rtwtypes.h` for the ERT target when both of the following conditions exist:

- The **Classic call interface** option on the **Code Generation** > **Interface** pane 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.

`rtwtypes.h` for GRT targets:

```
#ifndef __RTWTYPES_H__
#define __RTWTYPES_H__
#include "tmwtypes.h"
#include "simstruc_types.h"
#ifndef POINTER_T
# define POINTER_T

typedef void * pointer_T;

#endif

#ifndef TRUE
# define TRUE (1)
#endif

#ifndef FALSE
# define FALSE (0)
#endif
#endif
```

Top of `rtwtypes.h` for ERT targets:

```
#ifndef __RTWTYPES_H__
#define __RTWTYPES_H__
#ifndef __TMWTYPES__
#define __TMWTYPES__

#include <limits.h>

/*=======================================================================*
 * Target hardware information
 *   Device type: 32-bit Generic
 *   Number of bits:     char:   8    short:   16    int:  32
 *                       long:  32      native word size:  32
 *   Byte ordering: Unspecified
 *   Signed integer division rounds to: Undefined
 *   Shift right on a signed integer as arithmetic shift: on
 *=======================================================================*/

/*=======================================================================*
 * Fixed width word size data types:                                     *
 *   int8_T, int16_T, int32_T     - signed 8, 16, or 32 bit integers     *
 *   uint8_T, uint16_T, uint32_T  - unsigned 8, 16, or 32 bit integers   *
 *   real32_T, real64_T           - 32 and 64 bit floating point numbers *
```

```
 *=======================================================================*/

typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef short int16_T;
typedef unsigned short uint16_T;
typedef int int32_T;
typedef unsigned int uint32_T;
typedef float real32_T;
typedef double real64_T;
. . .
```

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 Simulink Coder build process places `rtwtypes.h` in `slprj/target/_sharedutils`; otherwise, it places `rtwtypes.h` in the standard build folder (*`model_target_`*`rtw`). See “Logging” on page 17-99 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 Simulink Coder type names or other Simulink Coder definitions. A typical example is for files that declare variables using a Simulink Coder data type, for example, `uint32_T myvar;`.

A source file that is intended to be used by the Simulink Coder product and by a Simulink S-function can leverage the preprocessor macro `MATLAB_MEX_FILE`. The macro is defined by the `mex` function:

```
#ifdef  MATLAB_MEX_FILE
#include "tmwtypes.h"
#else
#include "rtwtypes.h"
#endif
```

A source file meant to be used as the Simulink Coder `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.

For more information on the `rtwtypes.h` file, see “rtwtypes.h and Shared Utility Code”.

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);
  ```

A Simulink Coder target's `main.c` (or `.cpp`) file 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 filename `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 Simulink Coder Header Files, due to requirements of generated code.

System Header Files

Header File	Purpose	GRT Targets	ERT Targets
`<float.h>`	Defines math constants	Not included	Included when generated code honors the **Stop time** solver configuration parameter due to one of the following Simulink Coder interface option settings: • **MAT-file logging** selected • **Interface** set to `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 replacement library entry For more information, see "Choose a Code Replacement Library".
`<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 open the Configuration Parameters dialog box and select **Code Generation** > **Interface** > **MAT-file logging**. See "MAT-file logging".
`<stdlib.h>`	Provides utility functions such as `div()` and `abs()`	Included when the model includes • A Stateflow chart	Included when the model includes • A Stateflow chart and you select the **Support: floating-point numbers** Simulink

Header File	Purpose	GRT Targets	ERT Targets
		• A Math Function block configured for `mod()` or `rem()`, which generate calls to `div()`	Coder interface configuration parameter • A Math Function block configured for `mod()` or `rem()`, which generate calls to `div()`
`<string.h>`	Provides memory functions such as `memset()` and `memcpy()`	Included due to use of `memset()` in model initialization code	Included when block or model initialization code calls `memcpy()` or `memset()` For a list of relevant blocks, enter `showblockdatatypetable` in the MATLAB Command Window and look for blocks with the N2 note. To omit calls to `memset()` from model initialization code, select the **Remove root level I/O zero initialization** and **Remove internal data zero initialization** optimization configuration parameters.

Simulink Coder Header Files

Header File	Purpose	GRT Targets	ERT Targets
`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	Included when either of the following conditions exists:

Header File	Purpose	GRT Targets	ERT Targets
			• The model uses noninlined S-functions • You select the Simulink Coder interface configuration parameter **Classic call interface**
*model*_types.h	Defines model-specific data types	Included	Included
rt_logging.h	Supports MAT-file logging	Included	Included when you open the Configuration Parameters dialog box and select **Code Generation** > **Interface** > **MAT-file logging**. See "MAT-file logging".
rt_nonfinite.h	Provides support for nonfinite numbers in the generated code	Included	Included when you select one of the following Simulink Coder interface configuration parameters: • **MAT-file logging** • **Support non-finite numbers** (and the generated code requires nonfinite numbers)
rtw_continuous.h	Supports continuous time	Included by simstruc_types.h	Included when you select the Simulink Coder interface configuration parameter **Support: continuous time** and simstruc.h is not already included
rtw_extmode.h	Supports external mode	Included by simstruc_types.h	Included when you configure the model for external mode and simstruc.h is not already included
rtw_matlogging.h	Supports MAT-file logging	Included by simstruc_types.h	Included by rtw_logging.h

Header File	Purpose	GRT Targets	ERT Targets
		and `rtw_logging.h`	
`rtw_solver.h`	Supports continuous states	Included by `simstruc_types.h`	Included when you select the Simulink Coder interface configuration parameter **Support: floating-point numbers** and `simstruc.h` is not already included
`rtwtypes.h`	Defines Simulink Coder data types; generated file	Included; use the complete version of the file, which includes `tmwtypes.h` and `simstruc_types.h` (see `simstruc_types.h` for dependencies)	Included; use the complete or optimized version of the file as explained in "rtwtypes.h" on page 11-39
`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 on the **Code Generation > Interface** pane: • **Mat-file logging** • `External mode` from the **Interface** list	Included when multiword data types are used. Included when you select one or more of the following in the Configuration Parameters dialog box on the **Code Generation > Interface** pane: • **Mat-file logging** • `External mode` from the **Interface** list
`model_reference_types.`	Contains type definitions for timing bridges	Included for a model reference target or a model	Included for a model reference target or a model

Header File	Purpose	GRT Targets	ERT Targets
		containing model reference blocks	containing model reference blocks
`builtin_typeid_types.h`	Defines an enumerated type corresponding to built-in data types	Included when you select one or more of the following in the Configuration Parameters dialog box on the **Code Generation > Interface** pane: • **Mat-file logging** • `C API` from the **Interface** list	Included when you select one or more of the following in the Configuration Parameters dialog box on the **Code Generation > Interface** pane: • **Mat-file logging** • `C API` from the **Interface** list
`simstruc.h`	Provides support for calling noninlined S-functions that use the `Simstruct` definition; also includes `limits.h`, `string.h`, `tmwtypes.h`, and `simstruc_types.`	Included	Included when either of the following conditions exists: • The model uses noninlined S-functions. • You select the Simulink Coder interface configuration parameter **Classic call interface**.
`simstruc_types.h`	Provides definitions used by generated code and includes the header files `rtw_matlogging.` `rtw_extmode.h`, `rtw_continuous.` `rtw_solver.h`, and `sysran_types.h`	Included with `rtwtypes.h`	Not included; `rtwtypes.h` contains definitions and *`model`*`.h` contains header files

Header File	Purpose	GRT Targets	ERT Targets
`sysran_types.h`	Supports external mode	Included by `simstruc_types.h`	Included when you configure the model for external mode and `simstruc.h` is not already included

Note: Header file dependencies noted in the preceding 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.

Dependencies of the Generated Code

The Simulink Coder software can directly build standalone executables for the host system such as when using the GRT target. 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 Simulink Coder software 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 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 a number of reasons such as:

- Blocks in a model generate code that makes function calls. This 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 ANSI[4] 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.
- Simulink Coder options such as external mode, C API, and MAT-file logging are examples that trigger additional dependencies.
- Specifying custom code can introduce dependencies.

Providing the Dependencies

The Simulink Coder product provides several mechanisms for feeding file dependency information into the Simulink Coder 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

4. ANSI is a registered trademark of the American National Standards Institute, Inc.

- 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 filename is added to the list of sources to compile.
  - The S-Function block parameter `SFunctionModules` provides the ability to specify additional source filenames.
  - The `rtwmakecfg.m` mechanism provides further capability in specifying dependencies. See "Use rtwmakecfg.m API to Customize Generated Makefiles" on page 16-103 for more information.

  For more information on applying these approaches to legacy or custom code integration, see "Integrate External Code Using Legacy Code Tool".
- 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 provides the ability to 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

As previously mentioned, Simulink Coder targets are typically makefile based and the Simulink Coder product provides a "Template MakeFile (TMF) to makefile" conversion capability. 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 26-56 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 filename 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 manually determine 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 Simulink Coder and blockset source and header files is provided below to assist in locating the dependencies.

Simulink Coder Static File Dependencies

Several locations in the MATLAB folder tree contain static file dependencies specific to the Simulink Coder product:

- *`matlabroot`*`/rtw/c/src/`

  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.

- *matlabroot*/rtw/extern/include/*.h
- *matlabroot*/simulink/include/*.h

  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 above 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 leverage the `rtwmakecfg.m` mechanism to provide the Simulink Coder software 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 Simulink Coder 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. Additionally, you can specify compiler include paths with the `-I` compiler option. The Simulink Coder 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:

- Simulink Coder 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 filenames in the `#include` statements, and additionally leverage include paths.
- The generated code is compiled using the Simulink Coder build process.

  In this case, compiler include paths (`-I`) can be provided to the Simulink Coder 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 Simulink Coder build process. The include paths are propagated when the template makefile (TMF) is converted to the actual makefile.
  - When building a custom Simulink Coder target that is makefile-based, the desired include paths can be directly added into the targets template makefile.
  - A `USER_INCLUDES` make variable that specifies a folder in which the Simulink Coder build process should search for included files can be specified on the Simulink Coder `make` command. For example,

```
make_rtw USER_INCLUDES=-Id:\work\feature1
```

The user includes are passed to the command-line invocation of the make utility, which will add them to the overall flags passed to the compiler.

Recommended Approaches

The following are recommended approaches for using `#include` statements and include paths in conjunction with the Simulink Coder build process to generate code that remains portable and minimizes compatibility problems with future versions.

Assume that additional header files are located at

```
c:\work\feature1\foo.h
c:\work\feature2\bar.h
```

- A simple approach is to include in the `#include` statements only the filename, such as

  ```
  #include "foo.h"
  #include "bar.h"
  ```

  Then, the include path passed to the compiler should contain 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 Simulink Coder build process, avoid dependencies on its build and code generation folder structure, such as the *`model`*`_ert_rtw` build folder or the `slprj` code generation folder. Thus, the `#include` statements should not just be relative to where the generated source file exists. For example, if your MATLAB current working 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 Simulink Coder folder structure, you should instead use one of the approaches described above.

Files and Folders Created by Build Process

The following sections discuss

- "Files Created During the Build Process" on page 11-56
- "Folders Used During the Build Process" on page 11-61

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 very high-level programming language source file.

Note Files generated by the Embedded Coder build process are packaged slightly differently. Depending on model architectures and code generation options, the Simulink Coder build process might generate other files.

Descriptions of the principal generated files follow. Note that these descriptions use the generic term `model` for the model name:

- `model.rtw`

  An ASCII file, representing the compiled model, generated by the Simulink Coder build process. This file is analogous to the object file created from a high-level language source program. By default, the Simulink Coder 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 and is generated by the Target Language Compiler. This 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 "Header Dependencies When Interfacing Legacy/Custom Code with Generated Code" 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 Simulink Coder generate code only and skip the rest of the build process)
- *model*.mk

  Customized makefile generated by the Simulink Coder 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 Embedded Coder ERT targets that do not generate a GRT interface and do not have non-inlined 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 “Header Dependencies When Interfacing Legacy/Custom Code with Generated Code”.
- multiword_types.h

Contains type definitions for wide data types and their chunks. File is generated when multiword data types are used or when you select one or more of the following in the Configuration Parameters dialog box on the **Code Generation > Interface** pane:

  - **Mat-file logging**
  - `External mode` from the **Interface** list

- `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 you select one or more of the following in the Configuration Parameters dialog box on the **Code Generation > Interface** pane:

  - **Mat-file logging**
  - `C API` from the **Interface** list

- `rt_nonfinite.c`

  C source file that declares and initializes global nonfinite values for `inf`, minus `inf`, and `nan`. Nonfinite comparison functions are also provided. This file is generated for GRT-based targets and optionally generated for other targets.

- `rt_nonfinite.h`

  C header file that defines `extern` references to nonfinite variables and functions. This file is generated for GRT-based targets and optionally generated for other targets.

- `rtw_proj.tmw`

  Simulink Coder file generated for the `make` utility to use to determine when to rebuild objects when the name of the current Simulink Coder project changes

- *`model`*`.bat`

  Windows batch file used to set up the compiler environment and invoke the make command

- *`model`*`sources.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's signals, states, and parameters without using external mode. See "Data Interchange Using the C API" in Simulink Coder User's Guide for more information.
- `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 Simulink Coder software generates 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 above source files have dependency relationships, and there are additional file dependencies that you might need to take into account. These are described in "Generated Source Files and File Dependencies" in the Simulink Coder documentation.

Folders Used During the Build Process

The Simulink Coder build process places output files in three folders:

- The working folder

  If you choose to generate an executable program file, the Simulink Coder build process writes the file *`model`*`.exe` (Windows) or *`model`* (UNIX) to your working folder.

- The build folder — *`model_target`*`_rtw`

  A subfolder within your working 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).

- Code generation folder — `slprj`

  A subfolder within your working folder. When models referenced via Model blocks are built for simulation or Simulink Coder code generation, files are placed in `slprj`. The Embedded Coder configuration has an option that places generated shared code in `slprj` without the use of model reference. Subfolders in `slprj` provide separate places for simulation code, some Simulink Coder code, utility code shared between models, and other files. Of particular importance to Simulink Coder users are:

  - Header files from models referenced by this model

    `slprj/`*`target`*`/`*`model`*`/referenced_model_includes`

  - Model reference Simulink Coder target files

    `slprj/`*`target`*`/`*`model`*

  - MAT-files used during code generation of model reference Simulink Coder target and stand-alone code generation

    `slprj/`*`target`*`/`*`model`*`/tmwinternal`

  - Shared utilities

    `slprj/`*`target`*`/_sharedutils`

See “Work with Code Generation Folders” on page 4-14 for more information on organizing your files with respect to 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`)
- Simulink Coder code generation (`model.tmw`) files

For additional information about using code generation folders, see “Code Generation Folder Structure for Model Reference Targets” and “Customize Build to Use Shared Utility Code” in the Simulink Coder documentation.

How Code Is Generated From a Model

In this section...
“Model Compilation” on page 11-63
“Code Generation” on page 11-63

Model Compilation

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”).

When Simulink completes this processing, it compiles an intermediate representation of the model. This intermediate 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”.

Code Generation

The Simulink Coder code generator uses the Target Language Compiler (TLC) and a supporting TLC function library to transform the intermediate 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 those available on the MATLAB path with the system target file browser or you can type the name of such a file on your system prior to 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 Simulink Coder software 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

Code Generation of Matrices and Arrays

In this section...
"Simulink Coder Matrix Parameters" on page 11-66
"Internal Data Storage for Complex Number Arrays" on page 11-68

MATLAB, Simulink, and Simulink Coder software 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 nth 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.	• Transposing the row-major data in your legacy C code into column-major format as a 1–D array. • Using a "GetSet Custom Storage Class". The `GetSet` custom storage class replaces reads from (get) and writes to (set) with user-specified `GetSet` functions. The user-specified `GetSet` function has a single index argument, requiring the function to process the argument and access the target row and column data element. Using the `GetSet` function impacts code efficiency.
Integrate legacy C code functions requiring row-major data with Simulink generated data.	Using the Legacy Code Tool option `convert2DMatrixToRowMajor` to create an S-function that integrates legacy code functions with Simulink generated data. See `legacy_code`. The generated code stores the data as a 1–D array. Using `convert2DMatrixToRowMajor` impacts code efficiency.

Simulink Coder Matrix Parameters

The compiled model file, *model*.rtw, represents matrices as strings in MATLAB syntax, without an implied storage format. This format allows you to copy the string out of an .rtw file, paste it into a MATLAB file, and have it recognized by MATLAB.

Simulink Coder software 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

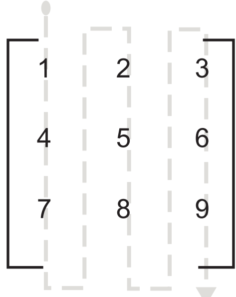
```
Parameter {
  Name   "OutputValues"
  Value   Matrix(3,3)
[[1.0, 2.0, 3.0]; [4.0, 5.0, 6.0]; [7.0, 8.0, 9.0];]
  String  "t"
  StringType   "Variable"
  ASTNode {
     IsNonTerminal  0
     Op  SL_NOT_INLINED
     ModelParameterIdx 3
    }
}
```

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 Simulink Coder 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 Simulink Coder, 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 (i.e., the signals do not actively copy their inputs, just the references).

Generated Code Considerations

In this section...
"Requirements for Signed Integer Representation" on page 11-69
"Subnormal Numbers" on page 11-69
"Specify Order of `rtwtypes.h` Include File" on page 11-70
"Default Folder for Code Generation" on page 11-70

Requirements for Signed Integer Representation

You must compile the code generated by Simulink Coder on a target using a two's complement representation for signed integer values. The software does not provide a check for this configuration in the generated code.

Subnormal Numbers

If the generated code performs calculations which produce or consume subnormal numbers, the execution of these calculations can be up to 50 times slower than similar calculations on normal numbers.

Subnormal numbers, formerly known as denormal numbers, fill the underflow gap around zero in floating-point arithmetic. Subnormal values are a special category of floating-point values which are too close to `0.0` to be represented as a normalized value. When the generated code adds and subtracts floating-point numbers, subnormal numbers prevent underflow.

To prevent target overruns:

- Set flush-to-zero mode on your processor to disable subnormal number support. In flush-to-zero mode, when used as input to a floating-point operation, a subnormal number is treated as 0. Underflow exceptions do not occur in flush-to-zero mode.
- Manually flush to zero any incoming or computed subnormal values at inputs and key operations such as washouts and filters.

To detect a subnormal value for single precision, 32–bit floating point numbers, do the following:

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 the range

```
 0 < fabsf(x) < SmallestNormalSingle
```

To detect a subnormal value for double precision, 64–bit floating point numbers do the following:

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 the range:

```
0 < fabs(x) < SmallestNormalDouble
```

For more information on floating-point and subnormal numbers, see the IEEE Standard 754, *IEEE Standard for Floating-Point Arithmetic*.

Specify Order of `rtwtypes.h` Include File

If code generation produces a program file which includes the header files `rtwtypes.h` and `tmwtypes.h`, then code generation includes `rtwtypes.h` before `tmwtypes.h`. This file order occurs whether the program file includes the header files directly or indirectly.

When code generation creates `rtwtypes.h`, it includes `typedef` definitions tailored for the target, using model parameter settings. `rtwtypes.h` is included first so that its target-specific `typedef` definitions are the definitions used when compiling the code.

For more information, see “rtwtypes.h and Shared Utility Code” on page 11-73.

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.

About Shared Utility Code

Blocks in a model can require common functionality to implement their algorithm. In many cases, it is most efficient to modularize this functionality into standalone support or helper functions, rather than inlining the code for the functionally 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 Simulink Coder software to generate code for your model.

In other cases, several model- and block-specific properties specify which functions are used and their behavior. Additionally, 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 statically define all possible function files before code generation. Instead, you can use the Simulink Coder shared utility mechanism, which generates support functions during code generation process.

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 of the Configuration Parameters dialog box. By default, the option is set to `Auto`. For this setting, if the model being built does not include Model blocks, the Simulink Coder build process places code required for fixed-point and other utilities in one of the following:

- The *`model`*`.c` or *`model`*`.cpp` file
- In a separate file in the Simulink Coder build folder (for example, `vdp_grt_rtw`)

Thus, the code is specific to the model.

If a model does contain Model blocks, the Simulink Coder build process 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`, where *`target`* is `sim` for simulations with Model blocks or the name of the system target file for Simulink Coder target builds. Some examples follow:

```
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 placements** option to `Shared location`. This forces the Simulink Coder build process to place utilities under the `slprj` folder rather than in the normal Simulink Coder build folder. This setting is useful when you are manually combining code from several models, as it prevents symbol collisions between the models.

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, you must then 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 **Code Generation > 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.
- Clear **Support: non-inlined S-functions**. If you use non-inlined S-functions in the model, this option generates an error.
- Clear **Classic call interface**. Disables use of the GRT interface.

Incremental Shared Utility Code Generation and Compilation

As explained in "Controlling Shared Utility Code Placement" on page 11-72, 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 the purpose of this discussion, the term functions means functions and macros.

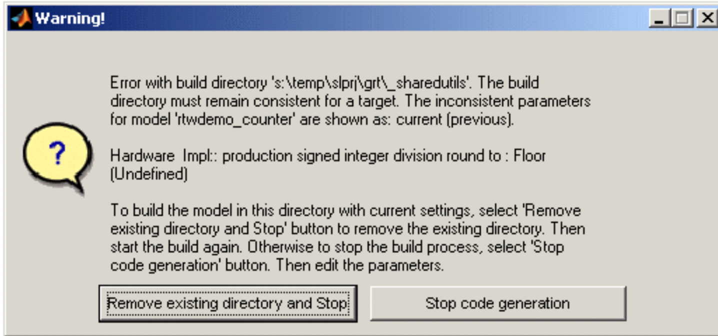
A shared function can be used by blocks within the same model and by blocks in different models when using model reference or when building multiple standalone models from the same start build folder. However, the Simulink Coder software generates the code for a given function only once for the block 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.

Shared Utility Checksum

As explained in "Incremental Shared Utility Code Generation and Compilation" on page 11-74, the Simulink Coder software uses the shared utilities folder when you explicitly configure a model to use the shared location or the model contains Model blocks. During the code generation process, if relative to the current folder, the configuration file `slprj/`*`target`*`/_sharedutils/checksummap.mat` exists, the product reads that file and makes sure that the current model being built has identical settings for the required model properties. If mismatches occur between the properties defined in `checksummap.mat` and the current model properties, a Warning dialog appears.



The following table lists properties that must match for the shared utility checksum.

Category	Properties

Hardware Implementation configuration properties	`get_param(bdroot, 'TargetShiftRightIntArith')`
	`get_param(bdroot, 'TargetEndianess')`
	`get_param(bdroot, 'ProdEndianess')`
	`get_param(bdroot, 'TargetBitPerChar')`
	`get_param(bdroot, 'TargetBitPerShort')`
	`get_param(bdroot, 'TargetBitPerInt')`
	`get_param(bdroot, 'TargetBitPerLong')`
	`get_param(bdroot, 'ProdHWWordLengths')`
	`get_param(bdroot, 'TargetWordSize')`
	`get_param(bdroot, 'ProdWordSize')`
	`get_param(bdroot, 'TargetHWDeviceType')`
	`get_param(bdroot, 'ProdHWDeviceType')`
	`get_param(bdroot, 'TargetIntDivRoundTo')`
	`get_param(bdroot, 'ProdIntDivRoundTo'`

Category	Properties

Additional configuration properties	`get_param(bdroot, 'TargetLibSuffix')`
	`get_param(bdroot, 'TargetLang')`
	`get_param(bdroot, 'TemplateMakefile')`
	`get_param(bdroot, 'CodeReplacementLibrary')`

ERT target properties	`get_param(bdroot, 'PurelyIntegerCode')` `get_param(bdroot, 'SupportNonInlinedSFcns'`
Platform property	Return value of the `computer` command

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	• 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	`get_param(bdroot, 'NoFixptDivByZeroProtection')`

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]
```

Below 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

Operation	FIX2FIX	FIX2FIX	ACCUM_POS	MUL	div	div
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.

Share User-Defined Data Types Across Models

In this section...

"About Sharing Data Types" on page 11-79
"Example: Sharing Simulink Data Type Objects" on page 11-80
"Example: Sharing Enumerated Data Types" on page 11-81

About Sharing Data Types

By default, user-defined data types that are shared among multiple models generate duplicate type definitions in the `model_types.h` file for each model. However, Simulink Coder software provides the ability to generate user-defined data type definitions into a header file in the `_sharedutils` folder that can be shared across multiple models, removing the need for duplicate copies of the data type definitions. 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 11-80 and "Example: Sharing Enumerated Data Types" on page 11-81.

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 will 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(Enumeration) 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, you must 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, you must 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(Enumeration) 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
```

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 will 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:

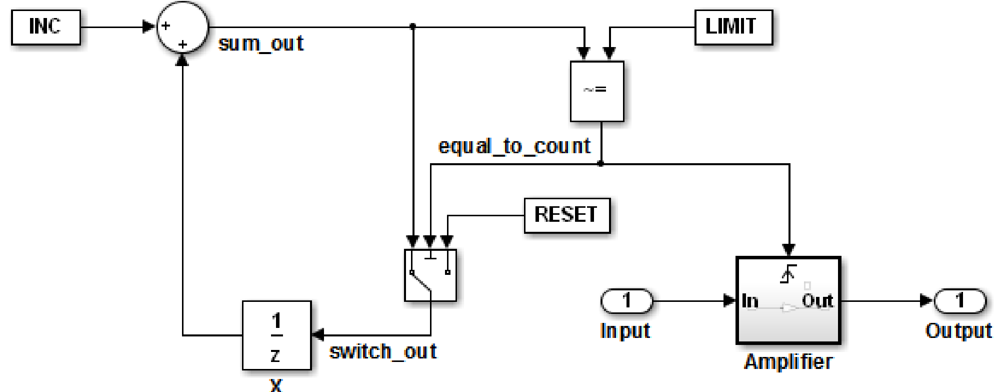
```
methods (Static = true)
...
     function retVal = getDataScope()
       retVal = 'Imported';
     end
...
end
```

Generating Code Using Simulink® Coder™

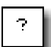
This example shows how to select a target for a Simulink® model, generate C code for real-time simulation, and view generated files.

1. Open the model.

```
model='rtwdemo_rtwintro';
open_system(model)
```



Algorithm Description
An 8-bit counter feeds a triggered subsystem parameterized by constants INC, LIMIT, and RESET. The I/O for the model is Input and Output. The Amplifier subsystem amplifies the input signal by gain factor K, which is updated whenever signal equal_to_count is true.



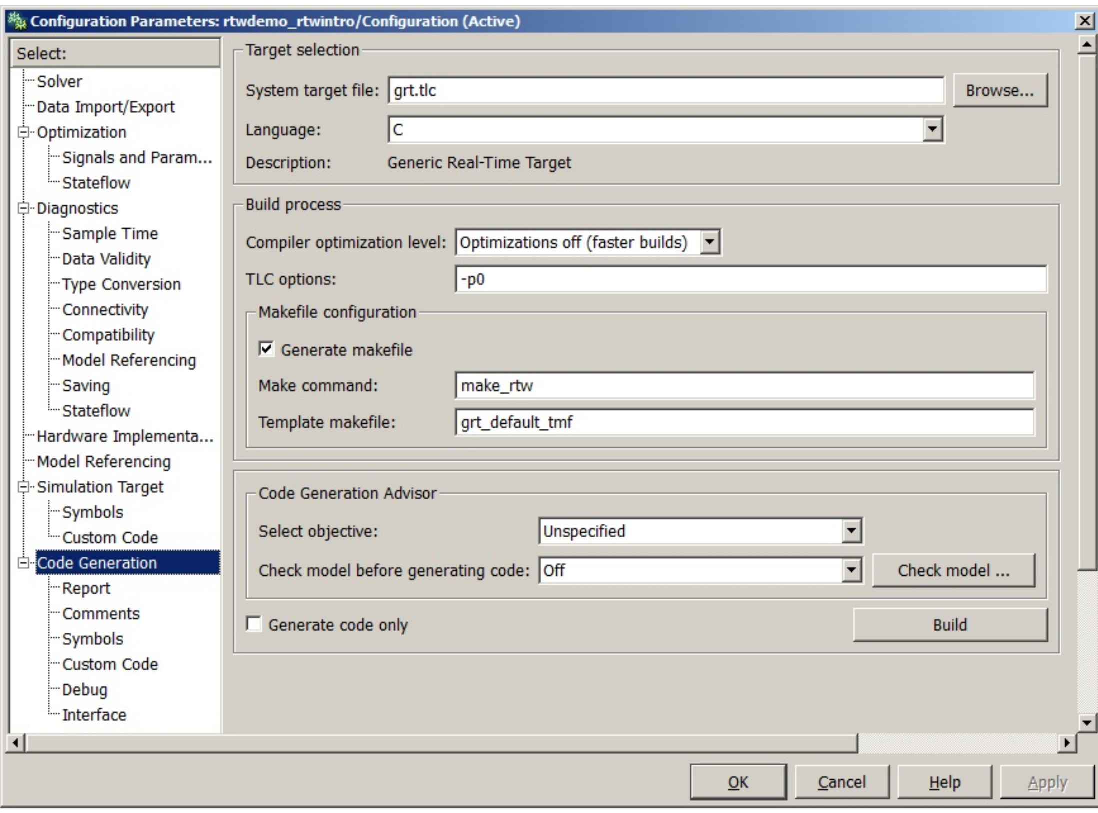
Copyright 1994-2012 The MathWorks, Inc.

2. Open the Configuration Parameters dialog box from the model editor by clicking **Simulation > Configuration Parameters**.

Alternately, type the following commands at the MATLAB® command prompt

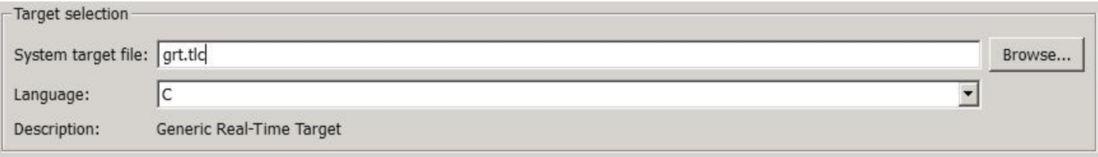
```
cs = getActiveConfigSet(model);
openDialog(cs);
```

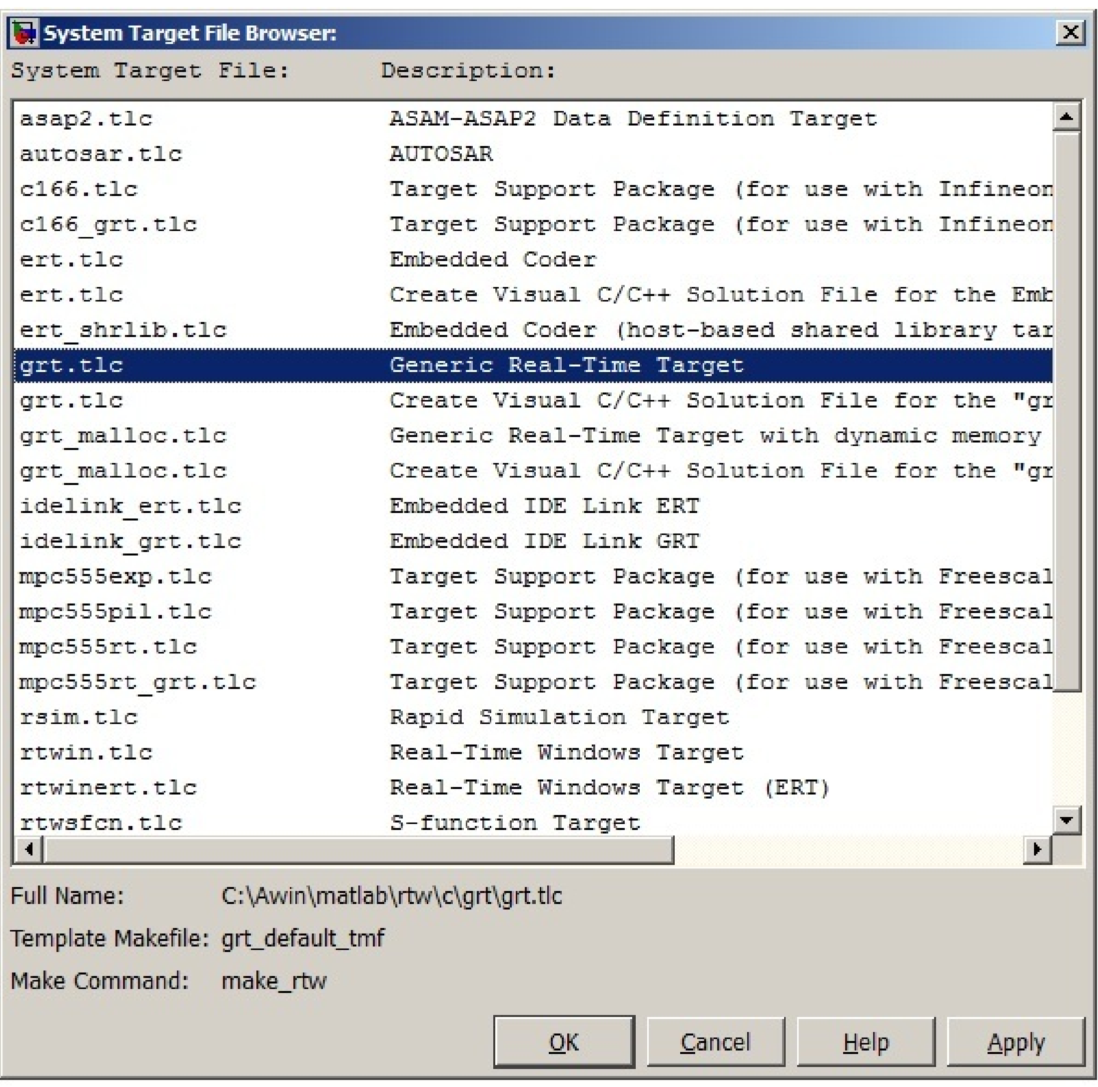
3. Select the **Code Generation** node.



4. In the **Target Selection** pane, click **Browse** to select a target.

You can generate code for a particular target environment or purpose. Some built-in targeting options are provided using system target files, which control the code generation process for a target.





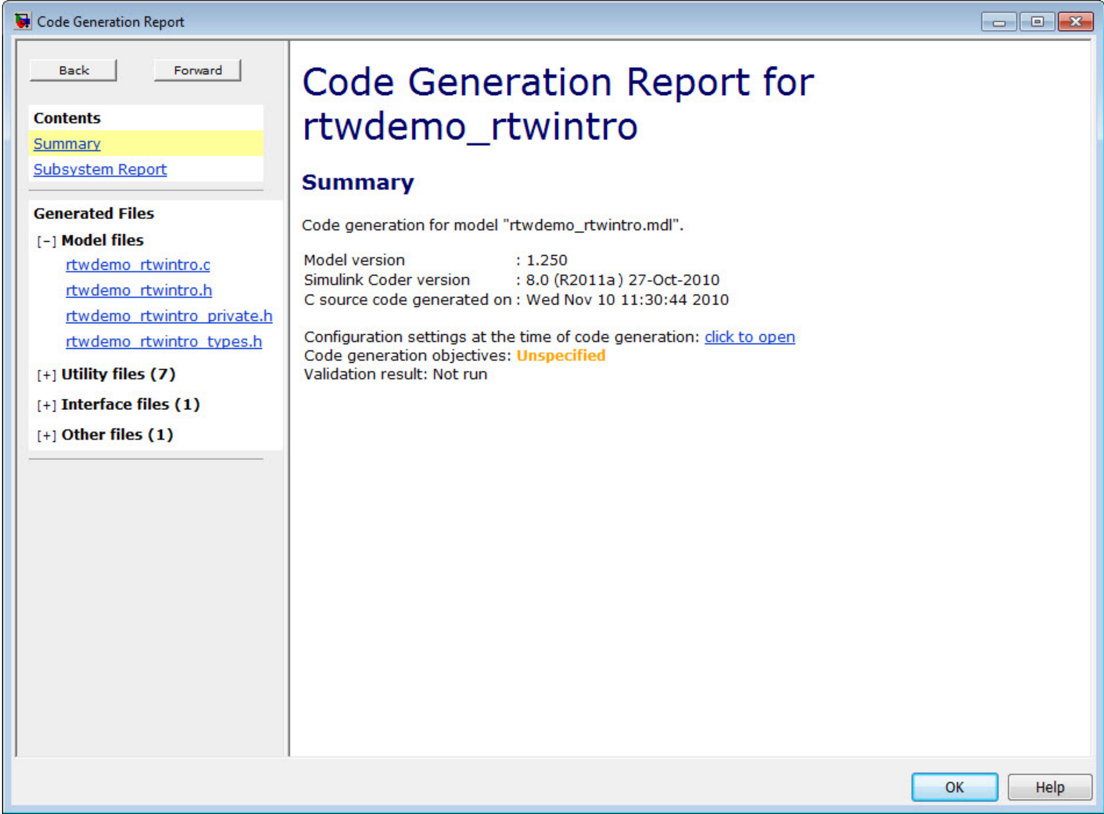
5. Select the **Generic Real-Time (GRT)** target and click **Apply**.

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 **Code Generation** pane, click **Build** to generate code.

7. View the code generation report that appears.

The report includes links to model files such as `rtwdemo_rtwintro.c` and associated utility and header files.



The figure below contains a portion of `rtwdemo_rtwintro.c`

Step function for model: rtwdemo_rtwintro

File: rtwdemo_rtwintro.c

```
1   /* Model step function */
2   void rtwdemo_rtwintro_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_rtwintro_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_rtwintro_PrevZCSigState.Amplifier_Trig_ZCE
22         != POS_ZCSIG)) {
23      /* Outport: '<Root>/Output' incorporates:
24       *  Gain: '<S1>/Gain'
25       *  Inport: '<Root>/Input'
26       */
27      rtwdemo_rtwintro_Y.Output = rtwdemo_rtwintro_U.Input << 1;
28    }
29
30    rtwdemo_rtwintro_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_rtwintro_DWork.X = rtb_sum_out;
39    } else {
40      /* Update for UnitDelay: '<Root>/X' incorporates:
41       *  Constant: '<Root>/RESET'
42       */
43      rtwdemo_rtwintro_DWork.X = 0U;
44    }
45
46    /* End of Switch: '<Root>/Switch' */
47  }
```

8. Close the model.

```
bdclose(model)
rtwdemoclean;
```

12

Report Generation

- “Reports for Code Generation” on page 12-2
- “HTML Code Generation Report Location” on page 12-3
- “HTML Code Generation Report for Referenced Models” on page 12-4
- “Generate a Code Generation Report” on page 12-5
- “Generate Code Generation Report After Build Process” on page 12-6
- “Open Code Generation Report” on page 12-8
- “Generate Code Generation Report Programmatically” on page 12-10
- “Search Code Generation Report” on page 12-11
- “View Code Generation Report in Model Explorer” on page 12-12
- “Package and Share the Code Generation Report” on page 12-14
- “Document Generated Code with Simulink Report Generator” on page 12-16

Reports for Code Generation

Simulink Coder software provides an HTML code generation report so that you can view and analyze the generated code. When your model is built, the code generation process produces an HTML file that is displayed in an HTML browser or in the Model Explorer. The code generation report includes:

- The **Summary** section lists version, date, and code generation objectives information. The **Configuration settings at the time of code generation** link opens a noneditable view of the Configuration Parameters dialog box. The dialog box shows the Simulink model settings at the time of code generation, including TLC options.
- The **Subsystem Report** section contains information on nonvirtual subsystems in the model.
- In the **Generated Files** section on the **Contents** pane, you can click the names of source code files generated from your model to view their contents in a MATLAB Web browser window. In the displayed source code, global variables are hypertext that links to their definitions.
- A **Find** box at the top of the window. For more information, see "Search Code Generation Report" on page 12-11.

For an example, see "Generate a Code Generation Report" on page 12-5 and "View Code Generation Report in Model Explorer" on page 12-12.

The contents of HTML reports varies depending on different target types. You can generate individual HTML reports for a subsystem or referenced model. For more information, see "HTML Code Generation Report for Referenced Models" on page 12-4 and "Generate Code for Referenced Models".

If you have a Simulink Report Generator license, you can document your code generation project in multiple formats, including HTML, PDF, RTF, Microsoft Word, and XML. For an example of how to create a Microsoft Word report, see "Document Generated Code with Simulink Report Generator" on page 12-16.

HTML Code Generation Report Location

The default location for the code generation report files is in the `html` subfolder of the build folder, *`model`*`_`*`target`*`_rtw/html/`. *`target`* is the name of the **System target file** specified on the **Code Generation** pane. The default name for the top-level HTML report file is *`model`*`_codegen_rpt.html` or *`subsystem`*`_codegen_rpt.html`. For more information on the location of the build folder, see "Control the Location for Generated Files".

HTML Code Generation Report for Referenced Models

To generate a code generation report for a top model and code generation reports for each referenced model, you need to specify the **Create code generation report** on the **Code Generation** > **Report** pane for the top model and each referenced model. You can open the code generation report of a referenced model in one of two ways:

- From the top-model code generation report, you can access the referenced model code generation report by clicking a link under **Referenced Models** in the left navigation pane. Clicking a link opens the code generation report for the referenced model in the browser. To navigate back to the top model code generation report, use the **Back** button at the top of the left navigation pane.
- From the referenced model diagram window, select **Code** > **C/C++ Code** > **Code Generation Report** > **Open Model Report**.

To generate a code generation report for a referenced model individually, follow the instructions in “Generate a Code Generation Report” on page 12-5 and “Open Code Generation Report” on page 12-8 for the referenced model.

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.

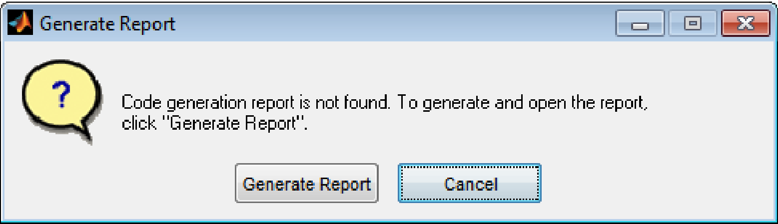
The build process writes the code generation report files to the `html` subfolder of the build folder (see “HTML Code Generation Report Location” on page 12-3). Next, the build process automatically opens a MATLAB Web browser window and displays the code generation report.

To open an HTML code generation report at any time after a build, see “Open Code Generation Report” on page 12-8 and “Generate Code Generation Report After Build Process” on page 12-6.

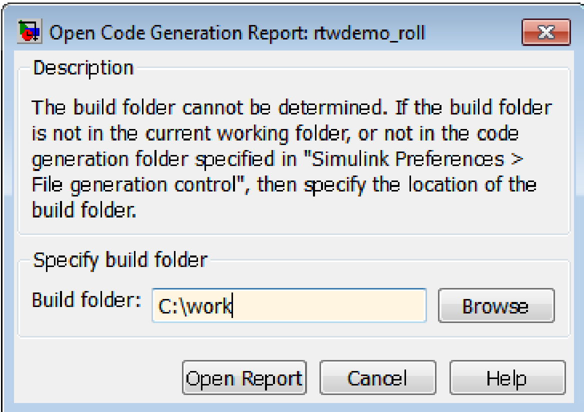
Generate Code Generation Report After Build Process

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**.

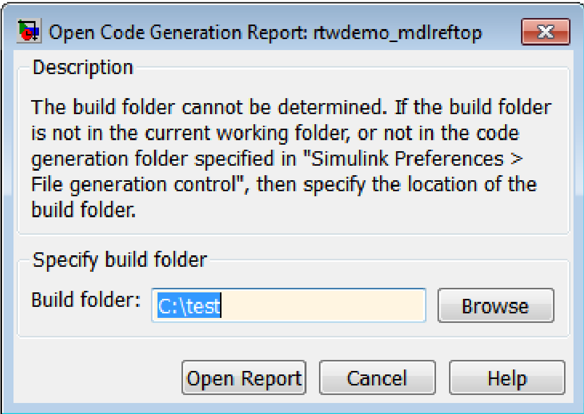
The software generates a report, `model_codgen_rpt.html`, from the code generation files in the build folder you specified.

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, you can open the report by selecting **Code** > **C/C++ Code** > **Code Generation Report** > **Open Model Report**. If you are opening a report for a subsystem, select **Open Subsystem Report**. A Simulink Coder license is required to view the code generation report. An Embedded Coder license is required to view a code generation report enhanced with Embedded Coder features.

If your current working folder does not contain the code generation files and the code generation report, the following dialog box opens:



Enter the full path of the build folder for your model, `../`*`model`*`_`*`target`*`_rtw` and click **Open Report**.

Alternatively, you can open the code generation report (*`model`*`_codegen_rpt.html` or *`subsystem`*`_codegen_rpt.html`) manually into a MATLAB Web browser window, or in another Web browser. For the location of the generated report files, see “HTML Code Generation Report Location” on page 12-3.

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.

Search Code Generation Report

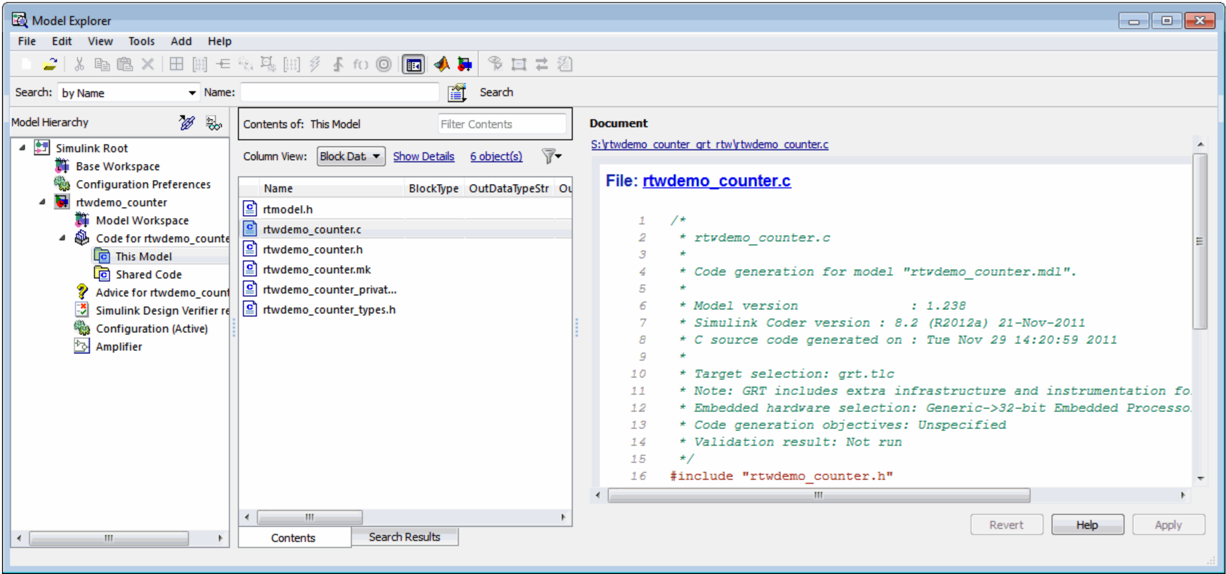
When the code generation report is displayed in the MATLAB Web browser window, you can search the report using the **Find** box at the top of the window. The search is not case sensitive.

Pressing **Ctrl-F** sets focus to the **Find** box. Type text into the **Find** box and hit **Enter** to start the search. The search highlights the found terms in the displayed page and scrolls to the first instance found. Press **Enter** to scroll through the subsequent search hits. If no terms are found, the background of the search box is highlighted red.

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 Simulink Coder software generates shared utility code for a model, a node named `Shared Code` appears directly under the **This Model** node. It collects source files that exist in the `./slprj/`*`target`*`/_sharedutils` subfolder.

Note You cannot use the **Search** tool built into Model Explorer toolbar to search generated code displayed in the Code Viewer. On PCs, typing **Ctrl+F** when focused on the **Document** pane opens a Find dialog box that you can use to search for strings in the currently displayed file. You can also search for text in the HTML report window, and you can open the files in the editor.

Package and Share the Code Generation Report

In this section...
"Package the Code Generation Report" on page 12-14
"View the Code Generation Report" on page 12-15

Package the Code Generation Report

To share the code generation report, you can package the code generation report files and supporting files into a zip file for transfer. The default location for the code generation report files is in two folders:

- `/slprj`
- `html` subfolder of the build folder, *`model_target`*`_rtw`, for example `rtwdemo_counter_grt_rtw/html`

To create a zip file from the MATLAB command window:

1. In the Current Folder browser, select the two folders:
   - `/slprj`
   - Build folder: *`model_target`*`_rtw`
2. Right-click to open the context menu.
3. In the context menu, select **Create Zip File**. A file appears in the Current Folder browser.
4. Name the zip file.

Alternatively, you can use the MATLAB `zip` command to zip the code generation report files:

```
zip('myzip',{'slprj','rtwdemo_counter_grt_rtw'})
```

Note: If you need to relocate the static and generated code files for a model to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB and Simulink products, use the Simulink Coder pack-and-go utility. For more information, see "Relocate Code to Another Development Environment".

View the Code Generation Report

To view the code generation report after transfer, unzip the file and save the two folders at the same folder level in the hierarchy. Navigate to the *model_target*_rtw/html/ folder and open the top-level HTML report file named *model*_codgen_rpt.html or *subsystem*_codegen_rpt.html in a Web browser.

Document Generated Code with Simulink Report Generator

In this section...

"Generate Code for the Model" on page 12-17
"Open the Report Generator" on page 12-17
"Set Report Name, Location, and Format" on page 12-19
"Include Models and Subsystems in a Report" on page 12-20
"Customize the Report" on page 12-21
"Generate the Report" on page 12-22

The Simulink Report Generator software creates documentation from your model in multiple formats, including HTML, PDF, RTF, Microsoft Word, and XML. This example shows one way to document a code generation project in Microsoft Word. The generated report includes:

- System snapshots (model and subsystem diagrams)
- Block execution order list
- Simulink Coder and model version information for generated code
- List of generated files
- Optimization configuration parameter settings
- Simulink Coder target selection and build process configuration parameter settings
- Subsystem map
- File name, path, and generated code listings for the source code

To adjust Simulink Report Generator settings to include custom code and then generate a report for a model, complete the following tasks:

1. "Generate Code for the Model" on page 12-17
2. "Open the Report Generator" on page 12-17
3. "Set Report Name, Location, and Format" on page 12-19
4. "Include Models and Subsystems in a Report" on page 12-20
5. "Customize the Report" on page 12-21
6. "Generate the Report" on page 12-22

A Simulink Report Generator license is required for the following report formats: PDF, RTF, Microsoft Word, and XML. For more information on generating reports in these formats, see the Simulink Report Generator documentation.

Generate Code for the Model

Before you use the Report Generator to document your project, generate code for the model.

1. In the MATLAB Current Folder browser, navigate to a folder where you have write access.
2. Create a working folder from the MATLAB command line by typing:

```
mkdir report_ex
```

3. Make `report_ex` your working folder:

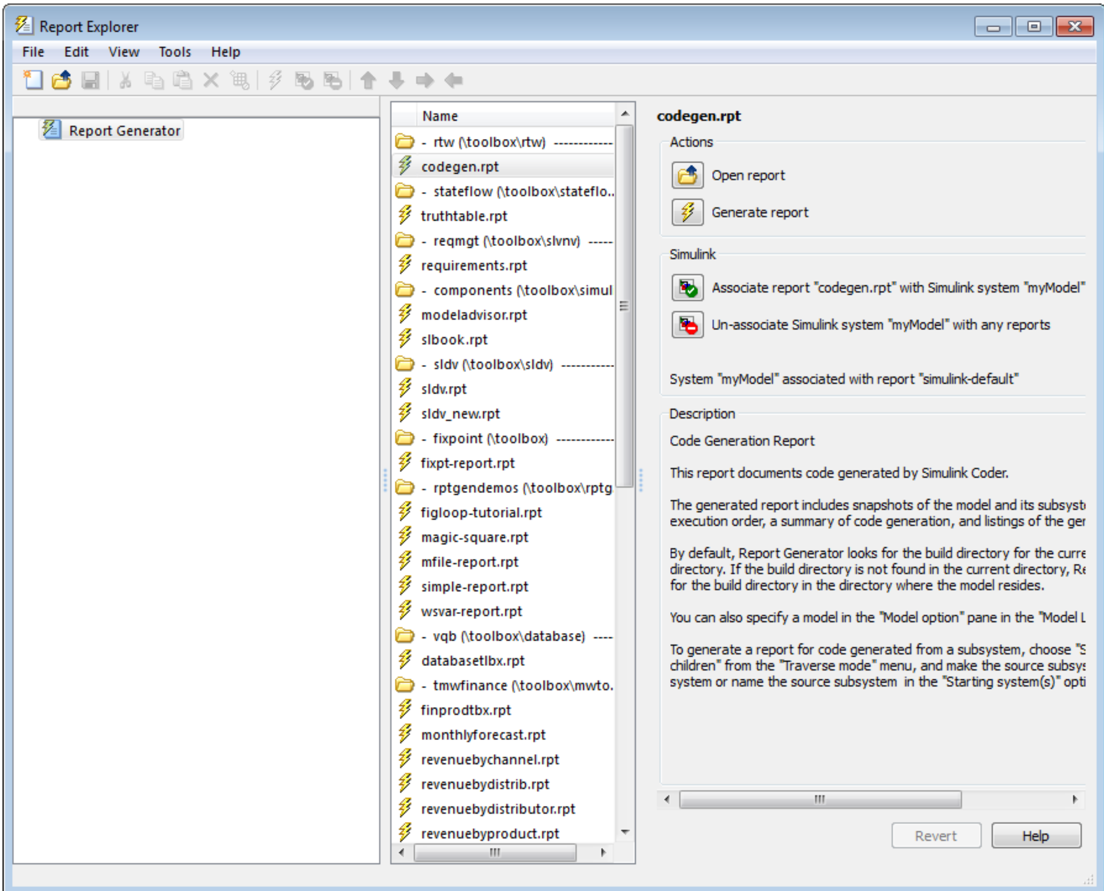
```
cd report_ex
```

4. Open the `slexAircraftExample` model by entering the model name on the MATLAB command line.
5. In the model window, choose **File > Save As**, navigate to the working folder, `report_ex`, and save a copy of the `slexAircraftExample` model as `myModel`.
6. Open the Configuration Parameters dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu.
7. Select the **Solver** pane. In the **Solver options** section, specify the **Type** parameter as `Fixed-step`.
8. Select the **Code Generation** pane. Select **Generate code only**.
9. Click **Apply**.
10. Click **Generate code**. The build process generates code for the model.

Open the Report Generator

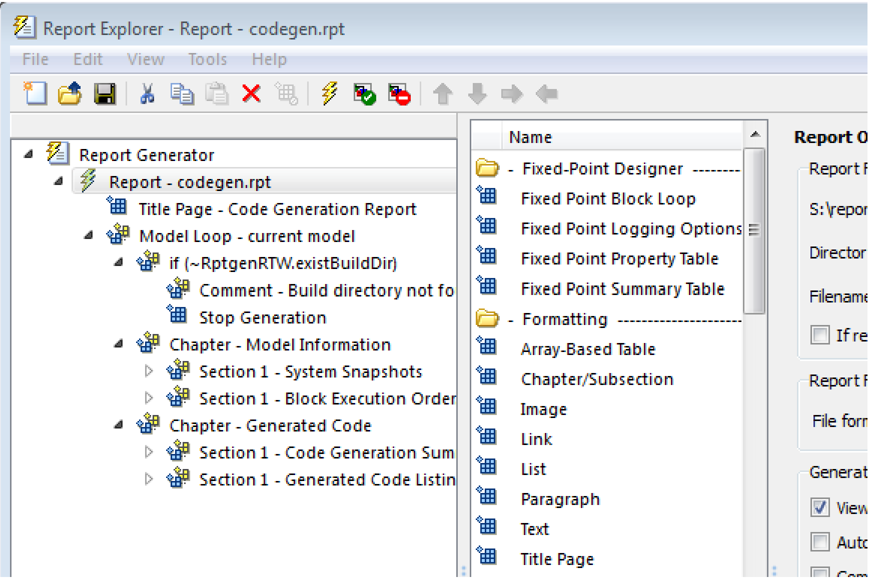
After you generate the code, open the Report Generator.

1. In the model diagram window, select **Tools** > **Report Generator**.
2. In the Report Explorer window, in the options pane (center), click the folder **rtw (\toolbox\rtw)**. Click the setup file that it contains, **codegen.rpt**.



3

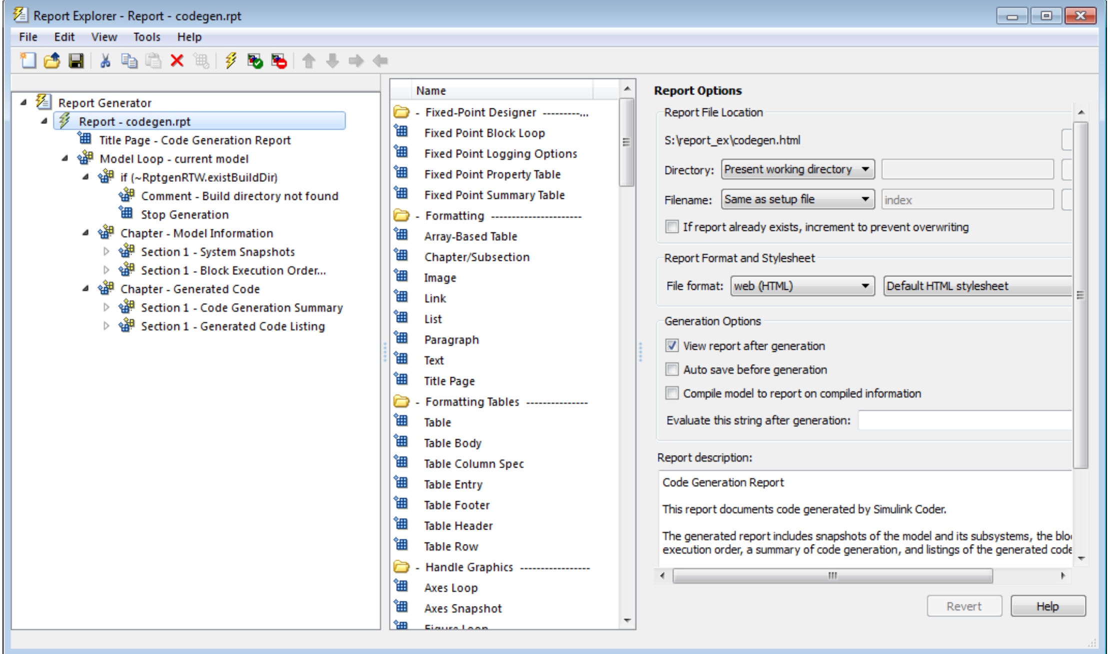
Double-click **codegen.rpt** or select it and click the **Open report** button . The Report Explorer displays the structure of the setup file in the outline pane (left).



Set Report Name, Location, and Format

Before generating a report, you can specify report output options, such as the folder, file name, and format. For example, to generate a Microsoft Word report named `MyCGModelReport.rtf`:

1 In the properties pane, under **Report Options**, review the options listed.



2 Leave the **Directory** field set to `Present working directory`.

3 For **Filename**, select `Custom:` and replace `index` with the name `MyModelCGReport`.

4 For **File format**, specify `Rich Text Format` and replace `Standard Print` with `Numbered Chapters & Sections`.

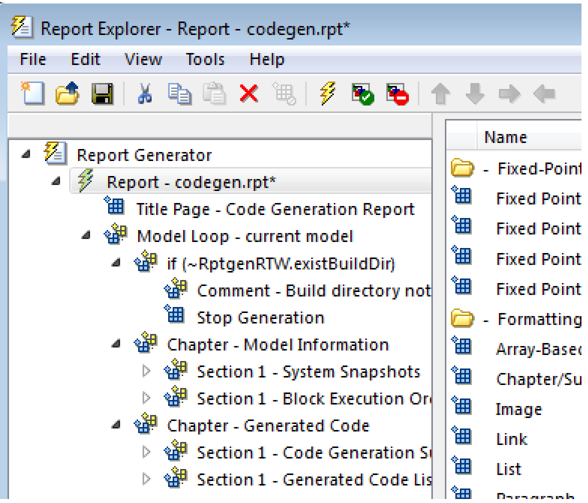
Include Models and Subsystems in a Report

Specify the models and subsystems that you want to include in the generated report by setting options in the Model Loop component.

1 In the outline pane (left), select **Model Loop**. Report Generator displays Model Loop component options in the properties pane.

2 If not already selected, select `Current block diagram` for the **Model name** option.

3 In the outline pane, click **Report - codegen.rpt***.



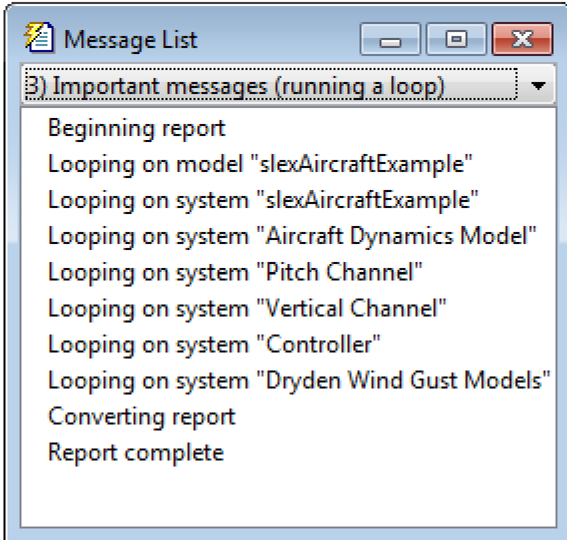
Customize the Report

After specifying the models and subsystems to include in the report, you can customize the sections included in the report.

1. In the outline pane (left), expand the node **Chapter - Generated Code**. By default, the report includes two sections, each containing one of two report components.
2. Expand the node **Section 1 — Code Generation Summary**.
3. Select **Code Generation Summary**. Options for the component are displayed in the properties pane.
4. Click **Help** to review the report customizations that you can make with the Code Generation Summary component. For this example, do not customize the component.
5. In the Report Explorer window, expand the node **Section 1 — Generated Code Listing**.
6. Select **Import Generated Code**. Options for the component are displayed in the properties pane.
7. Click **Help** to review the report customizations that you can make with the Import Generated Code component.

Generate the Report

After you adjust the report options, from the **Report Explorer** window, generate the report by clicking **File** > **Report**. A **Message List** dialog box opens, which displays messages that you can monitor as the report is generated. Model snapshots also appear during report generation. The **Message List** dialog box might be hidden behind other dialog boxes.



When the report is complete, open the report, `MyModelCGReport.rtf` in the folder `report_ex` (in this example).

For alternative ways of generating reports with the Simulink Report Generator, see “Generate Reports”.

Code Replacement for Simulink Models

- "What Is Code Replacement?" on page 13-2
- "Code Replacement Libraries" on page 13-4
- "Code Replacement Terminology" on page 13-6
- "Code Replacement Limitations" on page 13-9
- "Replace Code Generated from Simulink Models" on page 13-10
- "Choose a Code Replacement Library" on page 13-13

What Is Code Replacement?

Code replacement is a technique to change the code that the code generator produces for functions and operators to meet application code requirements. For example, you can replace generated code to meet requirements such as:

- Optimization for a specific run-time environment, including, but not limited to, specific target hardware.
- Integration with existing application code.
- Compliance with a standard, such as AUTOSAR.
- Modification of code behavior, such as enabling or disabling nonfinite or inline support.
- Application- or project-specific code requirements, such as:
  - Elimination of `math.h`.
  - Elimination of system header files.
  - Elimination of calls to `memcpy` or `memset`.
  - Use of BLAS.
  - Use of a specific BLAS.

To apply this technique, configure the code generator to apply a code replacement library (CRL) during code generation. By default, the code generator does not apply a code replacement library. You can choose from the following libraries that MathWorks provides:

- GNU C99 extensions—GNU[5] gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.
- Intel IPP for x86-64 (Windows)—Generates calls to the Intel® Performance Primitives (IPP) library for the x86-64 Windows platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Windows)—GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions.
- Intel IPP for x86/Pentium (Windows)—Generates calls to the Intel Performance Primitives (IPP) library for the x86/Pentium Windows platform.

5. GNU is a registered trademark of the Free Software Foundation.

- Intel IPP/SSE with GNU99 extensions for x86/Pentium (Windows)—Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86/Pentium Windows platform.
- Intel IPP for x86-64 (Linux)—Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Linux platform.
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)—Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86-64 Linux platform.

Libraries that include GNU99 extensions are intended for use with the GCC compiler. If use one of those libraries with another compiler, generated code might not compile.

Depending on the product licenses that you have, other libraries might be available . If you have an Embedded Coder license, you can view and choose from other libraries and you can create custom code replacement libraries.

Related Examples

- “Replace Code Generated from Simulink Models”
- “Choose a Code Replacement Library”

More About

- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Limitations”

Code Replacement Libraries

A *code replacement library* consists of one or more code replacement tables that specify application-specific implementations of functions and operators. For example, a library for a specific embedded processor specifies function and operator replacements that optimize generated code for that processor.

A *code replacement table* contains one or more *code replacement entries*, with each entry representing a potential replacement for a function or operator. Each entry maps a *conceptual representation* of a function or operator to an *implementation representation* and priority.

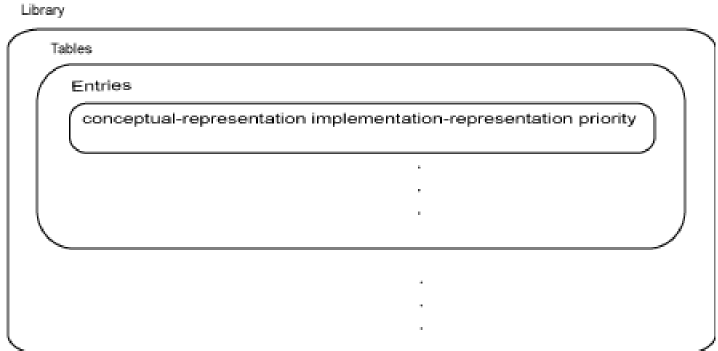


Table Entry Component	Description
Conceptual representation	Identifies the table entry and contains match criteria for the code generator. Consists of: • Function name or a key. The function name identifies most functions. For operators and some functions, a string called a key identifies a function or operator. For example, function name `'cos'` and operator key `'RTW_OP_ADD'`. • Conceptual arguments that observe code generator naming (`'y1'`, `'u1'`, `'u2'`, ...), with corresponding I/O types (output or input) and data types. • Other attributes, such as an algorithm, fixed-point saturation, and rounding modes, which identify matching criteria for the function or operator.

Table Entry Component	Description
Implementation representation	Specifies replacement code. Consists of: • Function name. For example, `'cos_dbl'` or `'u8_add_u8_u8'`) • Implementation arguments, with corresponding I/O types (output or input) and data types. • Parameters that provide additional implementation details, such as header and source file names and paths of build resources.
Priority	Defines the entry priority relative to other entries in the table. The value can range from 0 to 100, with 0 being the highest priority. If multiple entries have the same priority, the code generator uses the first match with that priority.

When the code generator looks for a match in a code replacement library, it creates and populates a *call site object* with the function or operator conceptual representation. If a match exists, the code generator uses the matched code replacement entry populated with the implementation representation and uses it to generate code.

The code generator searches the tables in a code replacement library for a match in the order that the tables appear in the library. If the code generator finds multiple matches within a table, the priority determines the match. The code generator uses a higher-priority entry over a similar entry with a lower priority.

Related Examples

- "Replace Code Generated from Simulink Models"
- "Choose a Code Replacement Library"

More About

- "What Is Code Replacement?"
- "Code Replacement Terminology"

Code Replacement Terminology

Term	Definition
Cache hit	A code replacement entry for a function or operator, defined in the specified code replacement library, for which the code generator finds a match.
Cache miss	A conceptual representation of a function or operator for which the code generator does not find a match.
Call site object	Conceptual representation of a function or operator that the code generator uses when it encounters a call site for a function or operator. The code generator uses the object to query the code replacement library for a conceptual representation match. If a match exists, the code generator returns a code replacement object, fully populated with the conceptual representation, implementation representation, and priority, and uses that object to generate replacement code.
Code replacement library	One or more code replacement tables that specify application-specific implementations of functions and operators. When configured to use a code replacement library, the code generator uses criteria defined in the library to search for matches. If a match is found, the code generator replaces code that it generates by default with application-specific code defined in the library.
Code replacement table	One or more code replacement table entries. Provides a way to group related or shared entries for use in different libraries.
Code replacement entry	Represents a potential replacement for a function or operator. Maps a conceptual representation of a function or operator to an implementation representation and priority.
Conceptual argument	Represents an input or output argument for a function or operator being replaced. Conceptual arguments observe naming conventions (`'y1'`,

Term	Definition
	'u1', 'u2', ...) and data types familiar to the code generator.
Conceptual representation	Represents match criteria that the code generator uses to qualify functions and operators for replacement. Consists of: • Function or operator name or key • Conceptual arguments with type, dimension, and complexity specification for inputs and output • Attributes, such as an algorithm and fixed-point saturation and rounding modes
Implementation argument	Represents an input or output argument for a C or C++ replacement function. Implementation arguments observe C/C++ name and data type specifications.
Implementation representation	Specifies C or C++ replacement function prototype. Consists of: • Function name (for example, 'cos_dbl' or 'u8_add_u8_u8') • Implementation arguments specifying type, type qualifiers, and complexity for the function inputs and output • Parameters that provide build information, such as header and source file names and paths of build resources and compile and link flags
Key	A string that identifies a function or operator that is being replaced. A function name or key appears in the conceptual representation of a code replacement entry. The key RTW_OP_ADD identifies the addition operator.

Term	Definition
Priority	Defines the match priority for a code replacement entry relative to other entries, which have the same name and conceptual argument list, within a code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If a library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

More About

- “What Is Code Replacement?”
- “Code Replacement Libraries”

Code Replacement Limitations

Code replacement verification — It is possible that code replacement behaves differently than you expect. For example, data types that you observe in code generator input might not match what the code generator uses as intermediate data types during an operation. Verify code replacements by examining generated code.

Related Examples

- “Replace Code Generated from Simulink Models”

More About

- “Code Replacement Libraries”

Replace Code Generated from Simulink Models

This example shows how to replace generated code, using a code replacement library. Code replacement is a technique you can use to change the code that the code generator produces for functions and operators to meet application code requirements.

Prepare for Code Replacement

1. Make sure that MATLAB, Simulink, Simulink Coder, and a C compiler are installed on your system. Some code replacement libraries available in your development environment can also require Embedded Coder.

   To install MathWorks products, see the MATLAB installation documentation. If you have installed MATLAB and want to see which other MathWorks products are installed, in the Command Window, enter `ver` .
2. Identify an existing or create a Simulink model for which you want the code generator to replace code.

Choose a Code Replacement Library

If you are not sure which library to use, explore the available libraries.

Configure Code Generator To Use Code Replacement Library

1. Configure the code generator to apply a code replacement library during code generation for the model.

   - In the Configuration Parameters dialog box, on the **Code Generation** > **Interface** pane, select a library from the **Code replacement library** menu.
   - Set the `CodeReplacementLibrary` parameter at the command line or programmatically.
2. Configure the code generator to produce code only (not build an executable) so you can verify your code replacements before building an executable.

   - In the Configuration Parameters dialog box, on the **Code Generation** pane, select **Generate code only**.
   - Set the `GenCodeOnly` parameter at the command line or programmatically.

Include Code Replacement Information In Code Generation Report

If you have an Embedded Coder license, you can configure the code generator to include a code replacement section in the code generation report. The additional information can help you verify code replacements.

1. Configure the code generator to generate a report. In the Configuration Parameters dialog box, on the **Code Generation** > **Report** pane, select **Create code generation report**. Consider having the report open automatically. Select **Open report automatically**.
2. Include the code replacement section in the report. On the **Code Generation** > **Report** pane, select **Summarize which blocks triggered code replacements**.

Generate Replacement Code

Generate C/C++ code from the model and, if you configured the code generator accordingly, a code generation report. For example, on the **Code Generation** > **General** pane, click **Generate Code**.

The code generator produces the code and displays the report.

Verify Code Replacements

Verify code replacements by examining the generated code. It is possible that code replacement behaves differently than you expect. For example, data types that you observe in the code generator input might not match what the code generator uses as intermediate data types during an operation.

Related Examples

- “Choose a Code Replacement Library”
- “Code Generation Configuration”

More About

- “Code replacement library”
- “Generate code only”
- “Create code generation report”
- “Open report automatically”
- “Summarize which blocks triggered code replacements”

- “What Is Code Replacement?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Limitations”

External Web Sites

- Supported Compilers

Choose a Code Replacement Library

In this section...
"About Choosing a Code Replacement Library" on page 13-13
"Explore Available Code Replacement Libraries" on page 13-13
"Explore Code Replacement Library Contents" on page 13-13

About Choosing a Code Replacement Library

By default, the code generator does not use a code replacement library.

If you are considering using a code replacement library:

1. Explore available libraries. Identify one that best meets your application needs.
   - Consider the lists of application code replacement requirements and libraries that MathWorks provides in "What Is Code Replacement?".
   - See "Explore Available Code Replacement Libraries".
2. Explore the contents of the library. See "Explore Code Replacement Library Contents".

If you do not find a suitable library and you have an Embedded Coder license, you can create a custom code replacement library. For more information, see "What Is Code Replacement Customization?" in the Embedded Coder documentation.

Explore Available Code Replacement Libraries

You can select the code replacement library to use for code generation on the "**Code Generation** > **Interface** pane " in the Configuration Parameters dialog box. To view a description of a library, select and hover your cursor over the library name. A tooltip describes the library and lists the tables that it contains. The tooltip lists the tables in the order that the code generator searches for a function or operator match.

Explore Code Replacement Library Contents

Use the Code Replacement Viewer to explore the content of a code replacement library.

1. At the command prompt, type `RTW.viewTfl`.

```
>> RTW.viewTfl
```

The viewer opens. To view the content of a specific library, specify the name of the library as an argument in single quotes. For example:

```
>> RTW.viewTfl('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.TflCOperationEntryGenerator` or `RTW.TflCOperationEntryGenerator_NetSlope`), the viewer displays an additional tab that shows fixed-point settings.

See Code Replacement Viewer for details on what the viewer displays.

Related Examples

- “Replace Code Generated from Simulink Models”

More About

- “What Is Code Replacement?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”
- “Code Replacement Limitations”

Deployment

14

Desktops

- “Rapid Simulations” on page 14-2
- “Generated S-Function Block” on page 14-31

Rapid Simulations

In this section...

"About Rapid Simulation" on page 14-2
"Rapid Simulation Advantage" on page 14-3
"General Rapid Simulation Workflow" on page 14-3
"Identify Rapid Simulation Requirements" on page 14-4
"Configure Inports to Provide Simulation Source Data" on page 14-6
"Configure and Build Model for Rapid Simulation" on page 14-6
"Set Up Rapid Simulation Input Data" on page 14-8
"Scripts for Batch and Monte Carlo Simulations" on page 14-18
"Run Rapid Simulations" on page 14-18
"Rapid Simulation Target Limitations" on page 14-30

About Rapid Simulation

After you create a model, you can use the Simulink Coder rapid simulation (RSim) target to characterize the model behavior. The RSim target executable that results from the build process is for non-real-time execution on your host computer. The executable 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 target ideal for batch or Monte Carlo simulations.

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)."

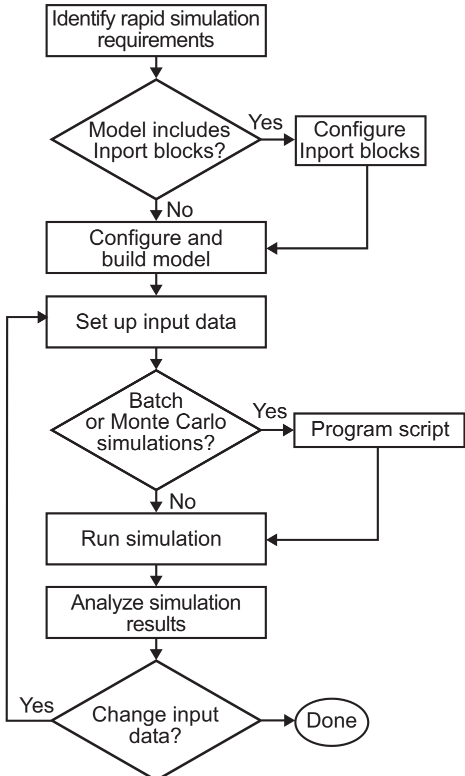
Rapid Simulation Advantage

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 14-6
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 14-6

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 14-6
Will you be running simulations with multiple data sets?	"Set Up Rapid Simulation Input Data" on page 14-8
Will the input data consist of global parameters, signals, or both?	"Set Up Rapid Simulation Input Data" on page 14-8
What type of source blocks provide input data to the model — From File, Inport, From Workspace?	"Set Up Rapid Simulation Input Data" on page 14-8
Will the model's parameter vector (`model_P`) be used as input data?	"Create a MAT-File That Includes a Model Parameter Structure" on page 14-9
What is the data type of the input parameters and signals?	"Set Up Rapid Simulation Input Data" on page 14-8
Will the source data consist of one variable or multiple variables?	"Set Up Rapid Simulation Input Data" on page 14-8
Does the input data include tunable parameters?	"Create a MAT-File That Includes a Model Parameter Structure" on page 14-9
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 14-9
Will you have a need to vary the simulation stop time for simulation runs?	"Configure and Build Model for Rapid Simulation" on page 14-6 and"Override a Model Simulation Stop Time" on page 14-21
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 14-21
Do you need to preserve the output of each simulation run?	"Specify a New Output File Name for a Simulation" on page 14-29 and "Specify New Output File Names for To File Blocks" on page 14-29

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 14-18

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 14-13. 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 Simulink Coder software populates the **Make command** and **Template makefile** fields with default settings and adds the **RSim Target** pane under **Code Generation**.

5 Click **RSim Target** to view the **RSim Target** pane.

Parameter loading

☑ Enable RSim executable to load parameters from a MAT-file

Solver

Solver selection: auto

Storage classes

☑ Force storage classes to AUTO

6 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 Simulink Coder software 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 Simulink Coder software saves simulation logging results to a file named `model.mat`. For more information, see "Export Simulation Data".

8 If you are using External mode communications, set up the interface, using the **Code Generation** > **Interface** pane. See "Host/Target Communication" on page 17-50 for details.

9 Return to the **Code Generation** pane and click **Build**. The Simulink Coder code generator builds a highly optimized executable that you can run on your host computer with varying data, without rebuilding.

   For more information on compilers that are compatible with the Simulink Coder product, see "Compiler or IDE Selection and Configuration" on page 17-2 and "Template Makefiles and Make Options" on page 10-35 .

Set Up Rapid Simulation Input Data

- "About Rapid Simulation Data Setup" on page 14-8
- "Create a MAT-File That Includes a Model Parameter Structure" on page 14-9
- "Create a MAT-File for a From File Block" on page 14-12
- "Create a MAT-File for an Inport Block" on page 14-13

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 (`model_P`)	Use the `rsimgetrtp` 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	In the Configuration Parameters dialog box, on the **Optimization** > **Signals and Parameters** pane, enable the **Inline Parameters** option. Call the `rsimgetrtp` 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.

If the Input Data Source Is...	Then...
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.
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
model	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
`modelChecksum`	A four-element vector that encodes the structure of the model. The Simulink Coder software uses the checksum to check whether

Field	Description
	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 *model*_P vector, the new checksum does not match the 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, `double`
`dataTypeID`	Internal data type identifier used by the Simulink Coder software
`complex`	The value 0 if real; 1 if complex
dtTransIdx	Internal data index used by Simulink Coder software
values	A vector of the parameter values associated with this structure
map	If you select the **Inline parameters** option, this field contains the mapping information that correlates the 'values' to the tunable parameters of the model. This mapping information, in conjunction with `rsimsetrtpparam`, is useful for creating subsequent rtP structures without compiling the block diagram.

If you select the **Inline parameters** option for the model, then tunable parameter information is also available in the parameters field.

The Simulink Coder software 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, `rsimgetrtp` returns the parameter structure for the example model rtwdemo_rsimtf to `param_struct`.

```
param_struct = rsimgetrtp('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 `rsimgetrtp`, 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 14-18.

Convert the Parameter Structure for Running Simulations on Varying Data Sets

If you need 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 by using the @ operator to specify the index for a specific parameter in the file.

To convert the structure to a cell array:

1. Save the `parameters` vector of the structure returned by `rsimgetrtp` to a temporary variable. The following example saves the parameter vector to temporary variable `p`.

   ```
   param_struct = rsimgetrtp('rtwdemo_rsimtf');
   p = param_struct.parameters;
   ```
2. Convert the structure to a cell array.

   ```
   param_struct.parameters = [];
   ```
3. Assign the saved contents of the temporary variable to the original structure name as an element of the cell array.

   ```
   param_struct.parameters{1} = p;
   param_struct.parameters{1}

   ans =

       dataTypeName: 'double'
         dataTypeId: 0
            complex: 0
         dtTransIdx: 0
             values: [-140 -4900 0 4900]
   ```

```
 map: []
```

4 Make a copy of the cell array to preserve the original parameter values.

```
param_struct.parameters{2} = param_struct.parameters{1};
param_struct.parameters{2}

ans =

    dataTypeName: 'double'
      dataTypeId: 0
         complex: 0
      dtTransIdx: 0
          values: [-140 -4900 0 4900]
 map: []
```

For a subsequent data set, increment the array index.

5 Modify a combination of the parameter values.

```
param_struct.parameters{2}.values=[-150 -5000 0 4950];
```

6 Repeat steps 4 and 5 for each parameter data set that you want to use as input to a rapid simulation of the model.

7 Save the cell array representing the parameter structure to a MAT-file.

```
save rtwdemo_rsimtf.mat param_struct;
```

For more information on how to specify each data set when you run the simulations, see "Change Block Parameters for an RSim Simulation" on page 14-27.

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	• Least flexible. • One variable. • Two or more *columns*. 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 `double`.

Format	Description
	For an example, see Single time/data matrix in the following procedure, step 4. For more information, see "Import Data Arrays" in the Simulink documentation.
Signal-and-time structure	• More flexible than the single time/data matrix format. • One variable. • Must contain two top-level fields: `time` and `signals`. The `time` field contains a *column* vector of the simulation times. The `signals` field contains an array of substructures, each of which corresponds to an Inport block. The substructure index corresponds to the Inport block number. Each `signals` substructure must contain a field named `values`. The `values` field must contain an array of inputs for the corresponding Inport block, where each input corresponds to a time point specified by the `time` field. • 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. For an example, see Signal-and-time structure in the following procedure, step 4. For more information on this format, see "Import Data Structures" in the Simulink documentation.

Format	Description
Per-port structure	• 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. For an example, see Per-port structure in the following procedure, step 4. For more information, see “Import Data Structures” in the Simulink documentation.

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 for the model. The Simulink Coder 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 14-18. For an example of a rapid simulation batch script, see the example `rtwdemo_rsim_batch_script`.

Run Rapid Simulations

- "Rapid Simulations" on page 14-19
- "Requirements for Running Rapid Simulations" on page 14-20
- "Set a Clock Time Limit for a Rapid Simulation" on page 14-21
- "Override a Model Simulation Stop Time" on page 14-21

- “Read the Parameter Vector into a Rapid Simulation” on page 14-22
- “Specify New Signal Data File for a From File Block” on page 14-22
- “Specify Signal Data File for an Inport Block” on page 14-25
- “Change Block Parameters for an RSim Simulation” on page 14-27
- “Specify a New Output File Name for a Simulation” on page 14-29
- “Specify New Output File Names for To File Blocks” on page 14-29

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_rsimtf
```

From the MATLAB command line, use

```
!rtwdemo_rsimtf
```

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	`model -f oldfilename.mat=newfilename.mat`
Print a summary of the options for RSim executable targets	`executable filename -h`
Read input data for an Inport block from a MAT-file	`model -i filename.mat`
Time out after *n* clock time seconds, where *n* is a positive integer value	`model -L n`
Write MAT-file logging data to file `filename.mat`	`model -o filename.mat`
Read a parameter vector from file `filename.mat`	`model -p filename.mat`

To...	Use...
Override the default TCP port (17725) for External mode	*model* -port *TCPport*
Write MAT-file logging data to a MAT-file other than the MAT-file used for the previous simulation	*model* -t *oldfilename*.mat=*newfilename*.mat
Run the simulation until the time value *stoptime* is reached	*model* -tf *stoptime*
Run in verbose mode	*model* -v
Wait for the Simulink engine to start the model in External mode	*model* -w

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, ($ARCH) is “win32”, whereas for a Linux machine, ($ARCH) is “glnx86”.
- 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/glnx86:$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_rsimtf -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 Simulink Coder 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_rsimtf -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 14-9. 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 14-12.

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.

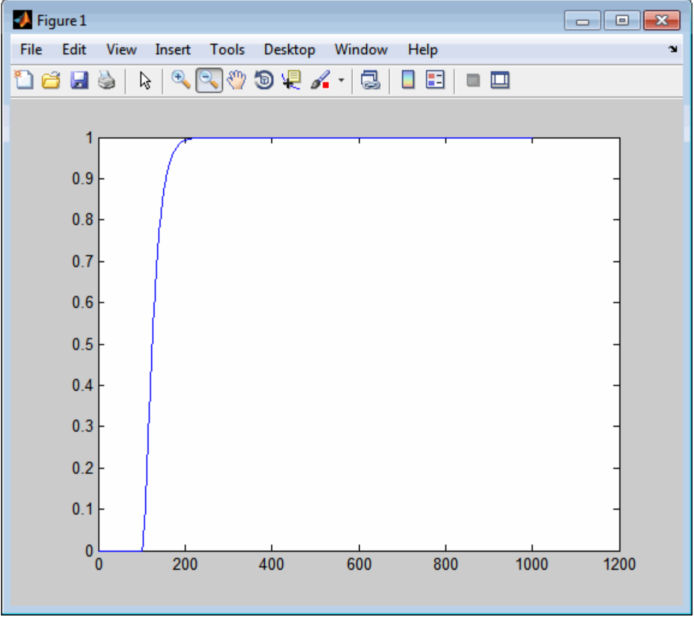
```
!rtwdemo_rsimtf
```

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;
```

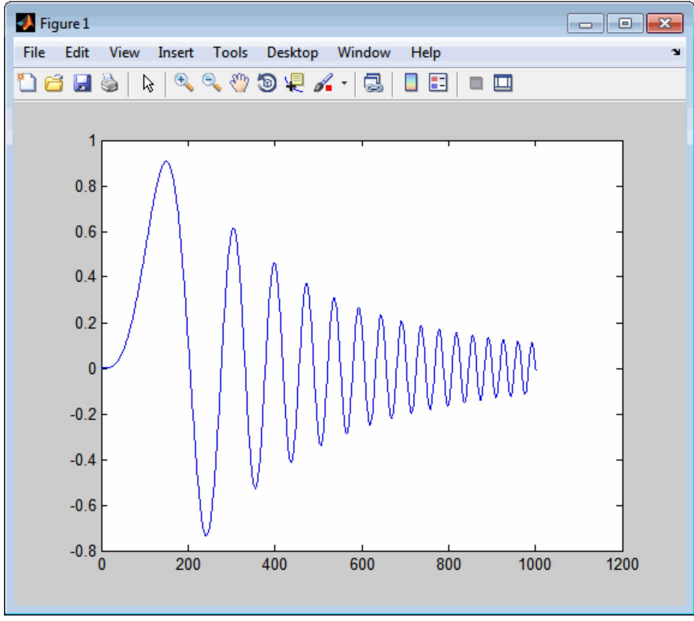
6 Run a rapid simulation with the new data by using the `-f` option to replace the original file, `rsim_tfdata.mat`, with `newfrom.mat`.

```
!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 14-13) 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 14-13.

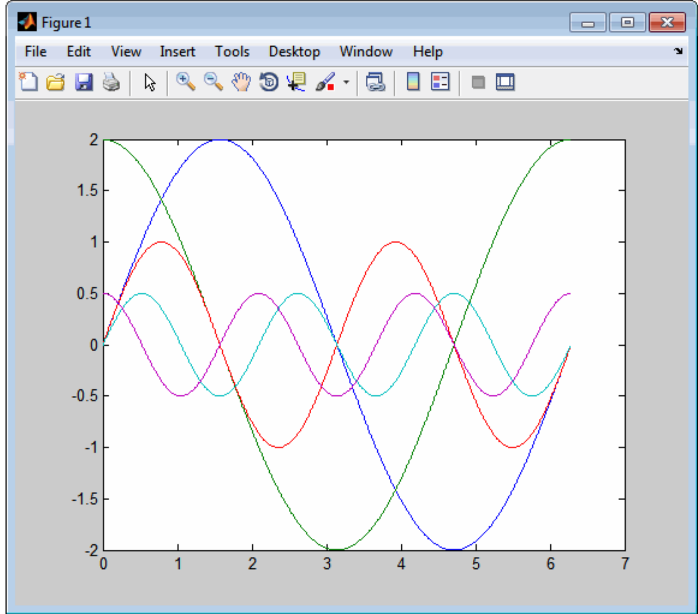
To specify the MAT-file after a simulation, you specify the executable with the `-i` option and the name of the MAT-file that contains the input data. For example:

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 14-6:

   - **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 14-13. 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`.

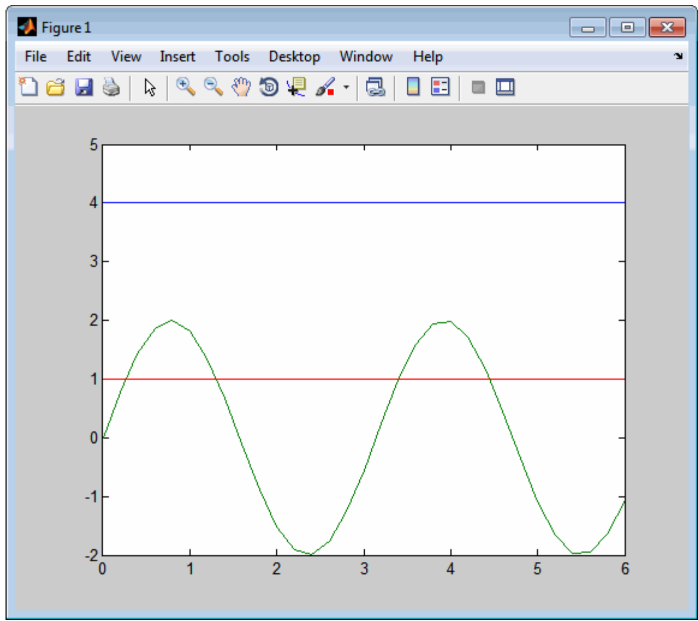
```
save rsim_i_single_struct.mat var_single_struct;
```

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 Dec 27 14:01:20 2005
 *** rsim_i_single_struct.mat is loaded! ***
** created rtwdemo_rsim_i.mat **

** Execution time = 0.2683734753333333sload rsim_i_single_struct.mat;
```

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 14-9, 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.

The RSim target allows you to alter model parameters, including those that contain *side-effects* functions. An example of a side-effects function is a simple Gain block that includes the following parameter entry in a dialog box:

```
gain value:   2 * a
```

The Simulink Coder code generator evaluates side-effects functions before generating code. The generated code for this example retains only one memory location entry, and the dependence on parameter `a` is not visible in the generated code anymore. The RSim target overcomes the problem of handling side-effects functions by replacing the entire parameter structure, *`model`*`_P`. You must create this new structure by using the `rsimgetrtp` function and then saving it in a MAT-file, as described in "Create a MAT-File That Includes a Model Parameter Structure" on page 14-9.

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 14-11, 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');
p = param_struct.parameters;
param_struct.parameters = [];
param_struct.parameters{1} = p;
```

```
param_struct.parameters{1}

ans =

    dataTypeName: 'double'
      dataTypeId: 0
         complex: 0
      dtTransIdx: 0
          values: [-140 -4900 0 4900]
param_struct.parameters{2} = param_struct.parameters{1};
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=mynewrsimdata.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 `mynewrsimdata.mat`. With this technique, you can avoid overwriting an existing simulation run.

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.
- Variable-step solver support for RSim is not available on Microsoft Windows platforms when you use the Watcom C/C++ compiler.

Generated S-Function Block

S-functions are an important class of target for which the Simulink Coder product can generate 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 Object Libraries" on page 14-31
"Create S-Function Blocks from a Subsystem" on page 14-34
"Tunable Parameters in Generated S-Functions" on page 14-38
"System Target File and Template Makefiles" on page 14-40
"Checksums and the S-Function Target" on page 14-41
"S-Function Target Limitations" on page 14-42

About Object Libraries

- "About the S-Function Target" on page 14-31
- "Required Files for S-Function Deployment" on page 14-32
- "Sample Time Propagation in Generated S-Functions" on page 14-33
- "Choose a Solver Type" on page 14-33

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 S-function code format 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 14-34 for an example of this procedure.

- Speeding up simulation. In many cases, 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 will continue to maintain its own unique data.

The S-function target generates noninlined S-functions. Within the same release, you can generate an executable from a model that contains generated S-functions by using the generic real-time target. This is not supported when incorporating a generated S-function from one release into a model that you build with a different release.

You can place a generated S-function block into another model from which you can generate another S-function. This allows any level of nested S-functions. For limitations related to nesting, see “Limitations on Nesting S-Functions” on page 14-46.

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 Embedded Coder 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 “Shared Object Libraries” in the Embedded Coder documentation.

Required Files for S-Function Deployment

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:
`*subsys*_sf.*mexext*`

where *subsys* is the subsystem name and *mexext* is a platform-dependent MEX-file extension (see “mexext”). For example, `SourceSubsys_sf.mexw32`.

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:

- `*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

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 "Inherit Sample Times" in the Simulink documentation and "Inherited Sample Time for Referenced Models" on page 4-26 in the Simulink Coder documentation.

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 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 tables show the possible combinations of source and destination model solver types for which the generated S-function works.

Subsystem with Continuous Blocks

If the Source Model Has a ...	Use this Solver Type in the Destination Model
Fixed-step solver	Fixed-step solver
Variable-step solver	Variable-step solver

Subsystem Without Continuous Blocks

If the Source Model Has a ...	Use this Solver Type in the Destination Model
Fixed-step solver	Fixed-step or variable-step solver
Variable-step solver	Variable-step solver

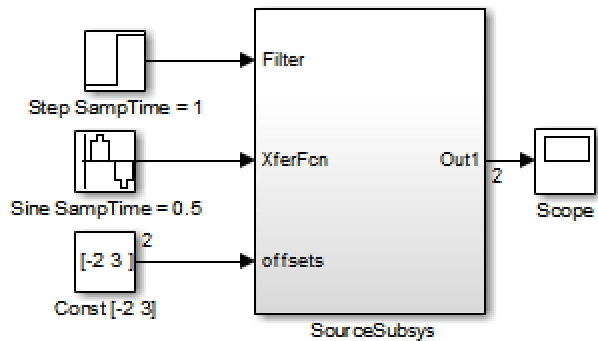
Note: S-functions generated from a subsystem have parameters that are hard coded into the block. Simulink calculates parameters such as sample time when it generates the block, not during simulation run time. Hence, it is important to verify whether the generated S-Function block works as expected in the destination model.

Create S-Function Blocks from a Subsystem

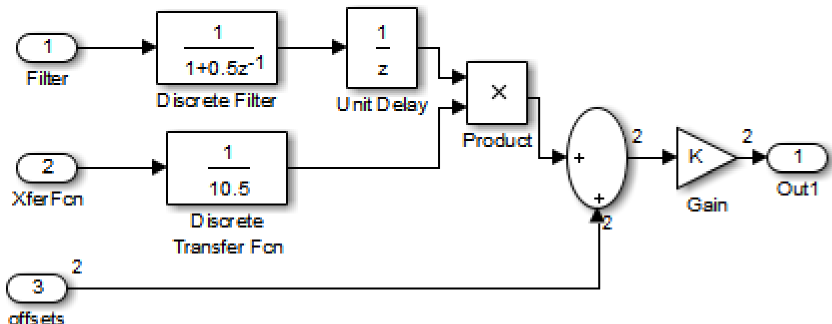
This section illustrates how to extract a subsystem from a model and generate a reusable S-function component from it.

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 will have 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 14-33.)

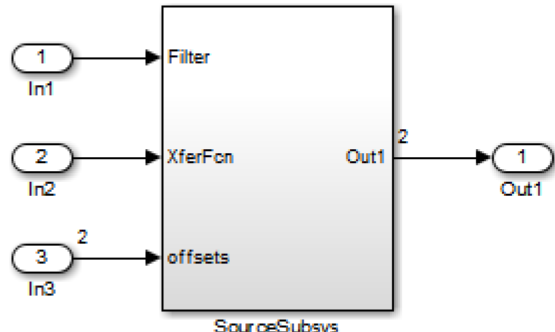
In this example, you want the S-Function block to retain the properties of `SourceSubsys` as it exists in `SourceModel`. Therefore, before you build the subsystem as a separate S-function component, you must set the inport sample times and widths explicitly. In addition, the solver parameters of the S-function component must be the same as those of the original model. The generated S-function component will operate identically to the original subsystem (see "Choose a Solver Type" on page 14-33 for an exception to this rule).

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 outport. Connect inports and an outport to `SourceSubsys`, as shown in the next figure.



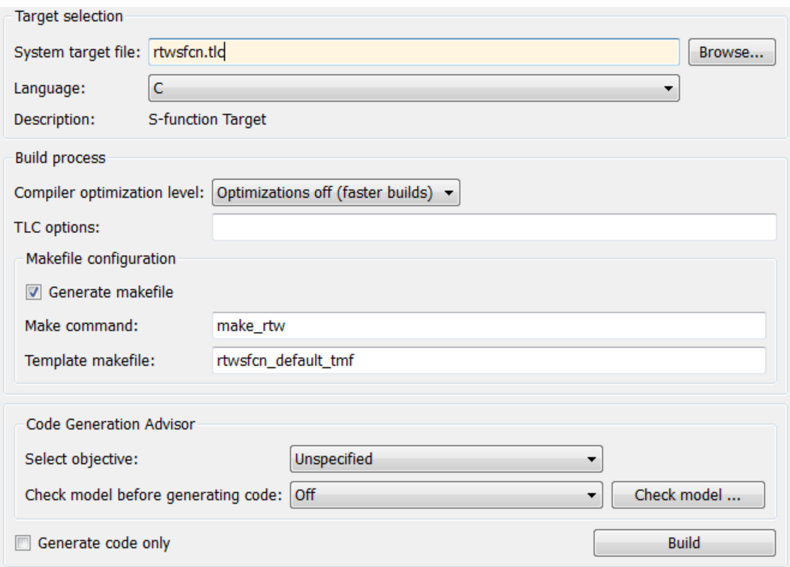
The signal widths and sample times are propagated to these ports.

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.

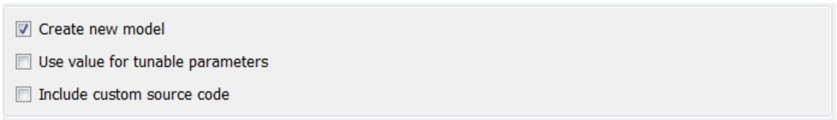
5 In the Configuration Parameters dialog box, go to the **Code Generation** pane.

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**. The **Code Generation** pane appears as follows.



8 Select the **S-Function Target** pane. Make sure that **Create new model** is selected, as shown in the next figure:



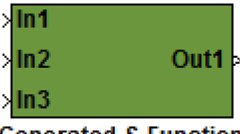
When this option is selected, the build process creates a new model after it builds the S-function component. The new model contains an S-Function block, linked to the S-function component.

Click **Apply**.

9 Save the new model containing your subsystem, for example as `SourceSubsys`.

10 Build the model.

11 The Simulink Coder build process builds the S-function component in the working folder. After the build, a new model window is displayed.

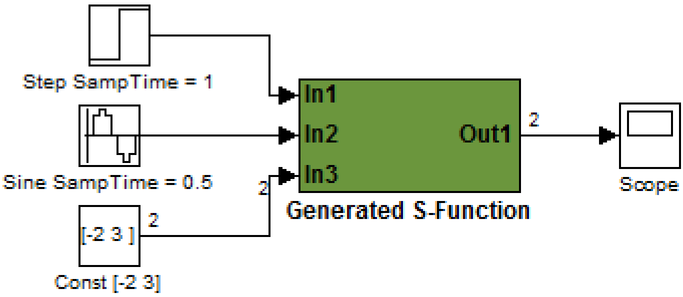


Optionally you can save the generated model, for example as `SourceSubsys_Sfunction`.

12 You can now copy the Simulink Coder S-Function block 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 14-32.

The next figure shows the S-Function block plugged into the original model. Given identical input signals, the S-Function block will perform 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" on page 16-21).

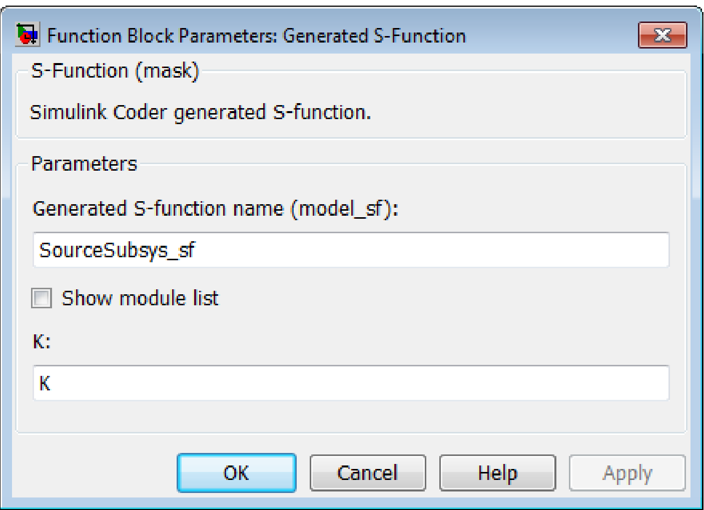
  or

- Use the Model Parameter Configuration dialog box (see "Parameters" on page 8-11) to declare desired block parameters tunable.

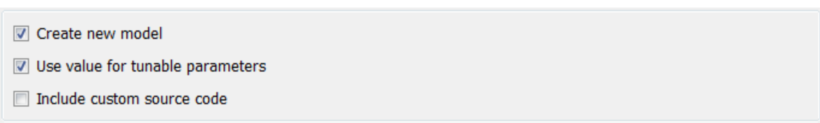
Block parameters that are declared tunable with the `auto` storage class in the source model become tunable parameters of the generated S-function. These parameters do not become part of a generated *`model`*`_P` (formerly `rtP`) parameter data structure, as they would in code generated from other targets. 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".

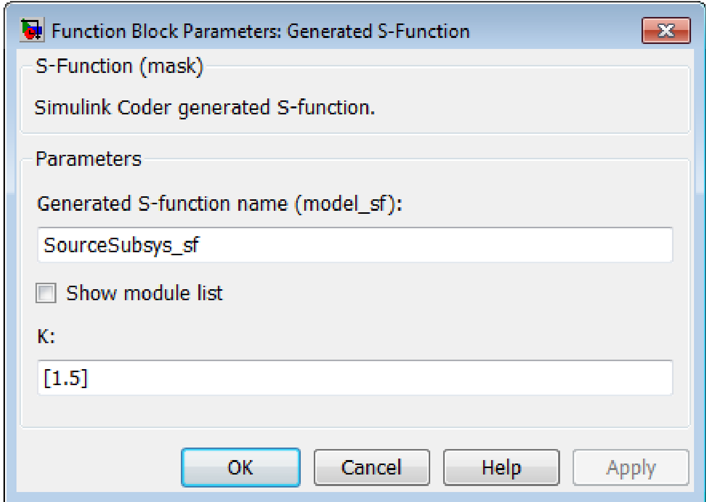
S-Function blocks created by using the S-function target are automatically masked. The mask displays each tunable parameter in an edit field. By default, the edit field displays the parameter by variable name, as in the following example.



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 and Template Makefiles

- "About System Target File and Template Makefiles" on page 14-40
- "System Target File" on page 14-41
- "Template Makefiles" on page 14-41

About System Target File and Template Makefiles

This section lists the target file and template makefiles that are provided for use with the S-function target.

System Target File

- `rtwsfcn.tlc`

Template Makefiles

- `rtwsfcn_lcc.tmf` — Lcc compiler
- `rtwsfcn_unix.tmf` — The Open Group UNIX host
- `rtwsfcn_vc.tmf` — Microsoft Visual C++ compiler
- `rtwsfcn_watc.tmf` — Watcom C compiler

Checksums and the S-Function Target

The Simulink Coder software creates a checksum for a Simulink model and uses the checksum during the build process for code reuse, model reference, and External mode features.

The Simulink Coder software calculates a model's 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 Simulink Coder software applies the subsystem and model checksums as follows:

- Code reuse — If two subsystems in a model have the same checksum, the Simulink Coder build process generates code for one function only.
- Model reference — If the current model checksum matches the checksum when the model was built, the Simulink Coder 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, the Simulink Coder build process generates an error.

S-Function Target Limitations

- "Limitations on Using Tunable Variables in Expressions" on page 14-42
- "Parameter Tuning" on page 14-42
- "Run-Time Parameters and S-Function Compatibility Diagnostics" on page 14-43
- "Limitations on Using Goto and From Block" on page 14-43
- "Limitations on Building and Updating S-Functions" on page 14-44
- "Unsupported Blocks" on page 14-45
- "`SimState` Not Supported for Code Generation" on page 14-45
- "Profiling Code Performance Not Supported" on page 14-45
- "Limitations on Nesting S-Functions" on page 14-46
- "Limitations on User-Defined Data Types" on page 14-46
- "Limitation on Right-Click Generation of an S-Function Target" on page 14-46
- "Limitation on S-Functions with Bus I/O Signals" on page 14-46
- "Limitation on Subsystems with Function-Call I/O Signals" on page 14-47
- "Data Store Access" on page 14-47
- "Cannot Specify Outport Dimensions Through Subsystem Mask" on page 14-47

Limitations on Using Tunable Variables in Expressions

Certain limitations apply to the use of tunable variables in expressions. When Simulink Coder software encounters an unsupported expression during code generation, 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 8-23.

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 (through the **Optimization** > **Signals and Parameters** > **Inline parameters** check box).
- 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 Simulink Coder software instructs you to upgrade S-functions that you create with the **Generate S-function** feature. This is because the S-function target 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`.

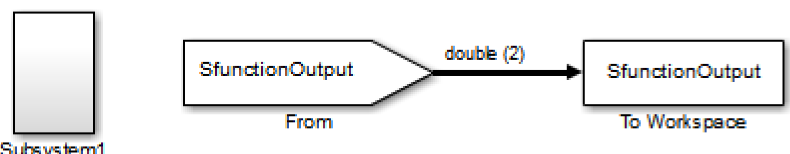
Limitations on Using Goto and From Block

When using the S-function target, the Simulink Coder code generator restricts I/O to correspond to the root model's 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.

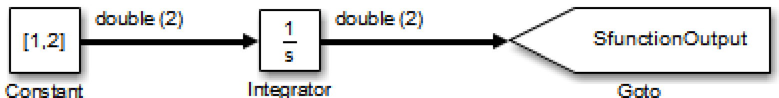
To work around this restriction, create your model and subsystem with the required Inport and Outport blocks, instead of using Goto and From blocks to pass data between the root model and subsystem. In the model that incorporates the generated S-function, you would then add Goto and From blocks.

Example Before Work Around

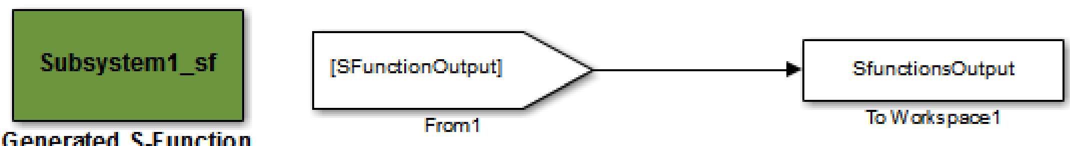
- Root model with a From block and subsystem, `Subsystem1`



- `Subsystem1` with a Goto block, which has global visibility and passes its input to the From block in the root model

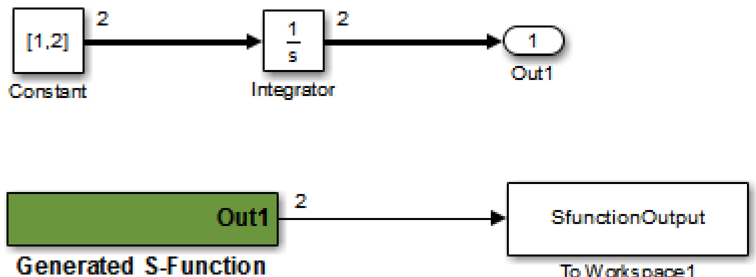


- `Subsystem1` replaced with an S-function generated with the S-Function target — a warning results when you run the model because the generated S-function does not implement the Goto block



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 Simulink Coder S-function target:

- You cannot build models that contain Model blocks using the Simulink Coder S-function target. 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 Simulink Coder 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 in the Embedded Coder product.

`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 Simulink Coder S-function target.

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 Simulink Coder S-function target 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, Simulink Coder S-function target code generation 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 Outport 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'
...
```

Real-Time Systems

- "Real-Time System Rapid Prototyping" on page 15-2
- "Hardware-In-the-Loop (HIL) Simulation" on page 15-5

Real-Time System Rapid Prototyping

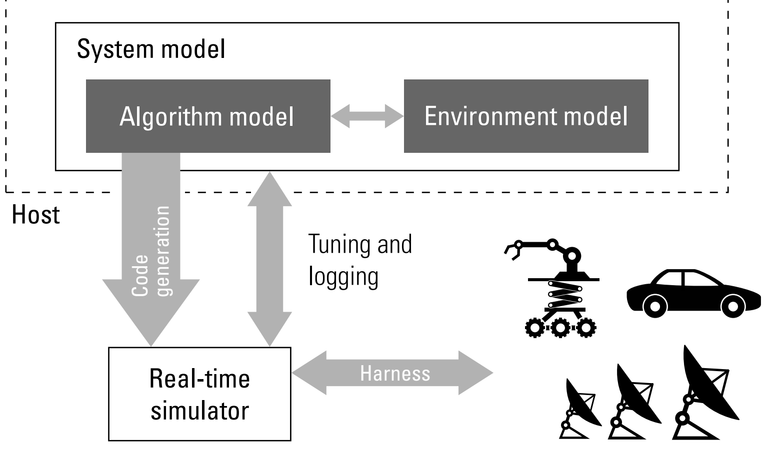
In this section...

"About Real-Time Rapid Prototyping" on page 15-2
"Goals of Real-Time Rapid Prototyping" on page 15-2
"Refine Code With Real-Time Rapid Prototyping" on page 15-3

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 Real-Time Windows Target 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 Simulink Coder 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	"Targets and Code Formats" in the Simulink Coder documentation Embedded Coder "What Are the Standards and Guidelines?" in the Embedded Coder documentation	`rtwdemo_counter` `rtwdemo_async`
Generate code for rapid prototyping in hard real time, using PCs	Simulink Real-Time	`help xpcdemos`

Engineering Tasks	Related Product Information	Examples
	"Configuration Parameters" in the Simulink Real-Time documentation	
Generate code for rapid prototyping in soft real time, using PCs	Real-Time Windows Target "Code Generation Pane: Real-Time Windows Target" in the Real-Time Windows Target documentation	`rtvdp` (and others)

3 Monitor signals, tune parameters, and log data.

Hardware-In-the-Loop (HIL) Simulation

In this section...

"About Hardware-In-the-Loop Simulation" on page 15-5
"Set Up and Run HIL Simulations" on page 15-6

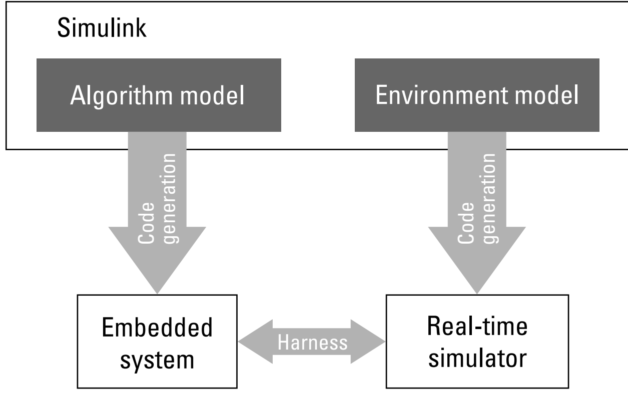
About Hardware-In-the-Loop Simulation

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 Simulink Coder product, you then generate, build, and download an executable 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:
   - "Targets and Code Formats"
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.

External Code Integration

- "Integration Options" on page 16-2
- "Reuse Algorithmic Components in Generated Code" on page 16-5
- "Deploy Algorithm Code Within a Target Environment" on page 16-12
- "Export Generated Algorithm Code for Embedded Applications" on page 16-16
- "Export Algorithm Executables for System Simulation" on page 16-19
- "Modify External Code for Language Compatibility" on page 16-20
- "Automate S-Function Generation" on page 16-21
- "Integrate External Code Using Legacy Code Tool" on page 16-25
- "Configure Model for External Code Integration" on page 16-29
- "Insert Custom Code Blocks" on page 16-32
- "Insert S-Function Code" on page 16-40

Integration Options

In this section...
"About Integration Options" on page 16-2
"Types of External Code Integration" on page 16-2

About Integration Options

The Simulink Coder product 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 must be integrated with code generated by the Simulink Coder software. *Custom code* is legacy code or other user-specified lines of code that must be included in the Simulink Coder build process. Collectively, legacy and custom code are called *external code*.

There are two ways that you can achieve external code integration. You can import existing external code into code generated 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.

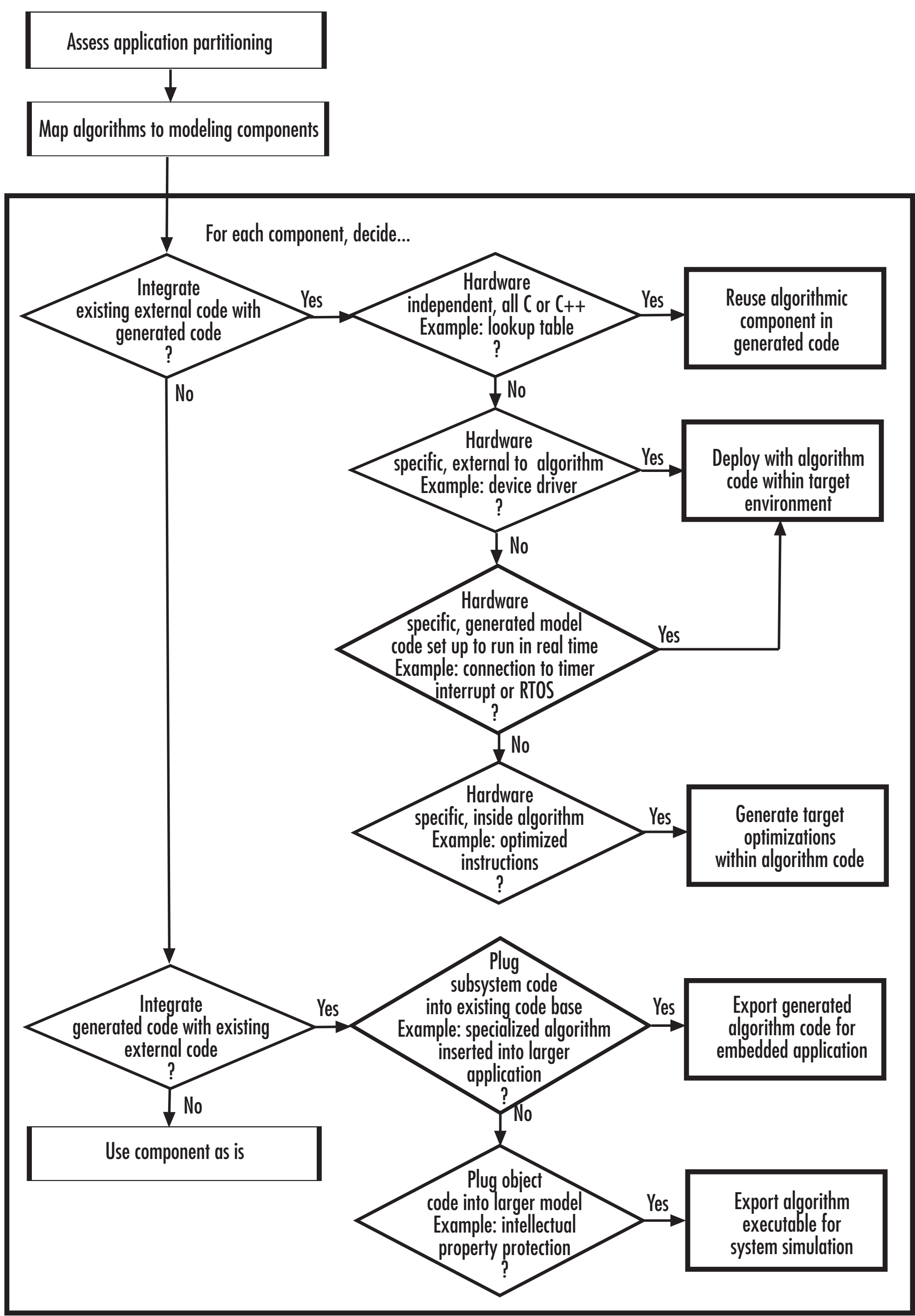
Types of External Code Integration

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.



Assess application partitioning
Map algorithms to modeling components
For each component, decide...
Integrate existing external code with generated code ?
Yes
No
Hardware independent, all C or C++ Example: lookup table ?
Yes
Reuse algorithmic component in generated code
No
Hardware specific, external to algorithm Example: device driver ?
Yes
Deploy with algorithm code within target environment
No
Hardware specific, generated model code set up to run in real time Example: connection to timer interrupt or RTOS ?
Yes
No
Hardware specific, inside algorithm Example: optimized instructions ?
Yes
Generate target optimizations within algorithm code
Integrate generated code with existing external code ?
Yes
No
Use component as is
Plug subsystem code into existing code base Example: specialized algorithm inserted into larger application ?
Yes
Export generated algorithm code for embedded application
No
Plug object code into larger model Example: intellectual property protection ?
Yes
Export algorithm executable for system simulation

Reuse Algorithmic Components in Generated Code

In this section...

"Reusable Algorithmic Components" on page 16-5
"Integrate External MATLAB Code" on page 16-5
"Integrate External C or C++ Code" on page 16-7
"Integrate Fortran Code" on page 16-10
"Integration Considerations for Reusable Algorithmic Components" on page 16-10

Reusable Algorithmic Components

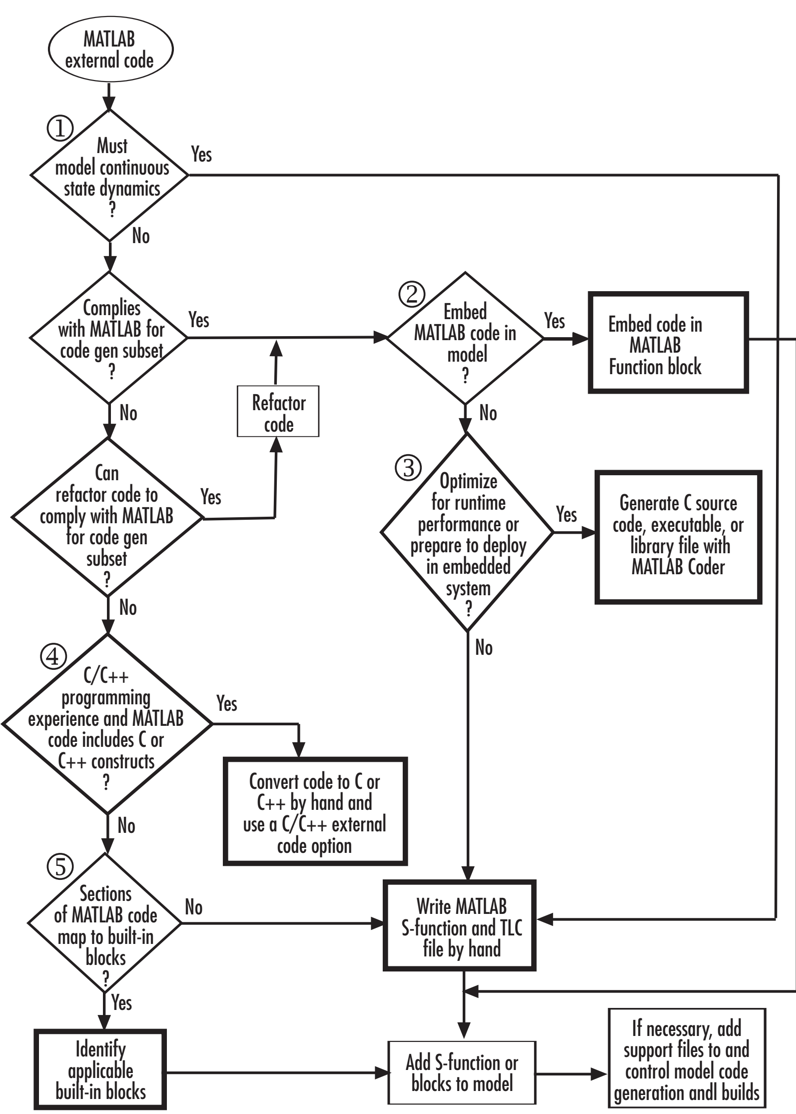
You have several options for integrating reusable algorithmic components with the generated code. Such components are hardware-independent and can be verified with Simulink simulation. Examples of such components include:

- 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.



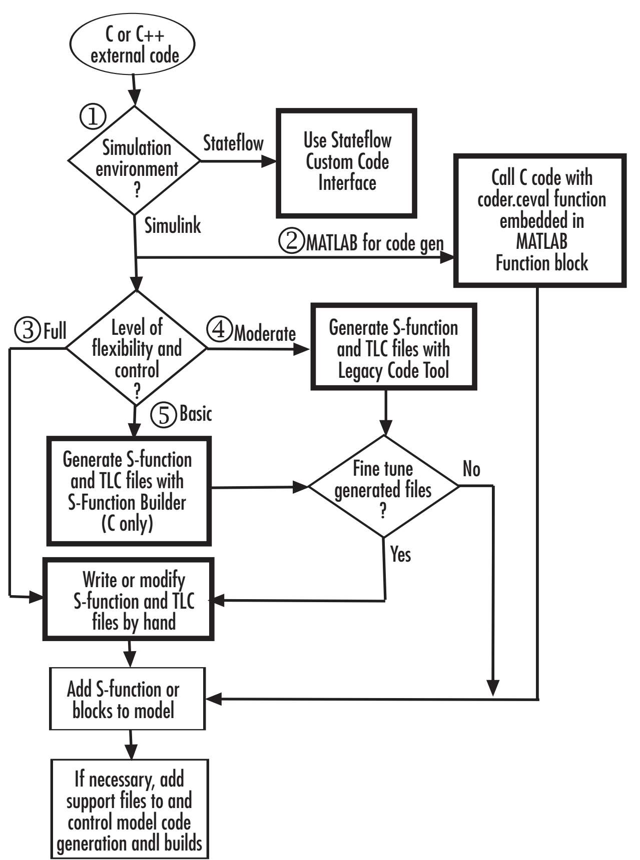
	If...	Then...	For More Information, See...
1	The algorithm must model continuous state dynamics	Write a MATLAB S-function and, for generating code, a corresponding TLC	• “MATLAB S-Functions” • “Inline MATLAB File S-Functions”

	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 16-7
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	• "Modeling Basics" and "Component-Based Modeling" • "Supported Products and Block Usage"

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	• `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 `coder.ceval` function from a MATLAB Function block	• `coder.ceval` function description • MATLAB Function block
3	You want maximum flexibility and the	Write an S-function and TLC file manually	• “C S-Function Examples” and “C++ S-Function Examples”

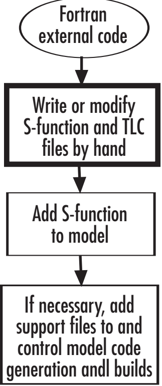
	If...	Then...	For More Information, See...
	ability to control what code is generated; the application requires function overloading or you need to format data definitions such that they are compatible with a function		• "S-Function Basics" • "Insert S-Function Code" on page 16-40
4	You want ease of 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	• `rtwdemo_lct_lut_script` • "Integrate C Functions Using Legacy Code Tool" in the Simulink documentation • "Integrate External Code Using Legacy Code Tool"
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`, and `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.

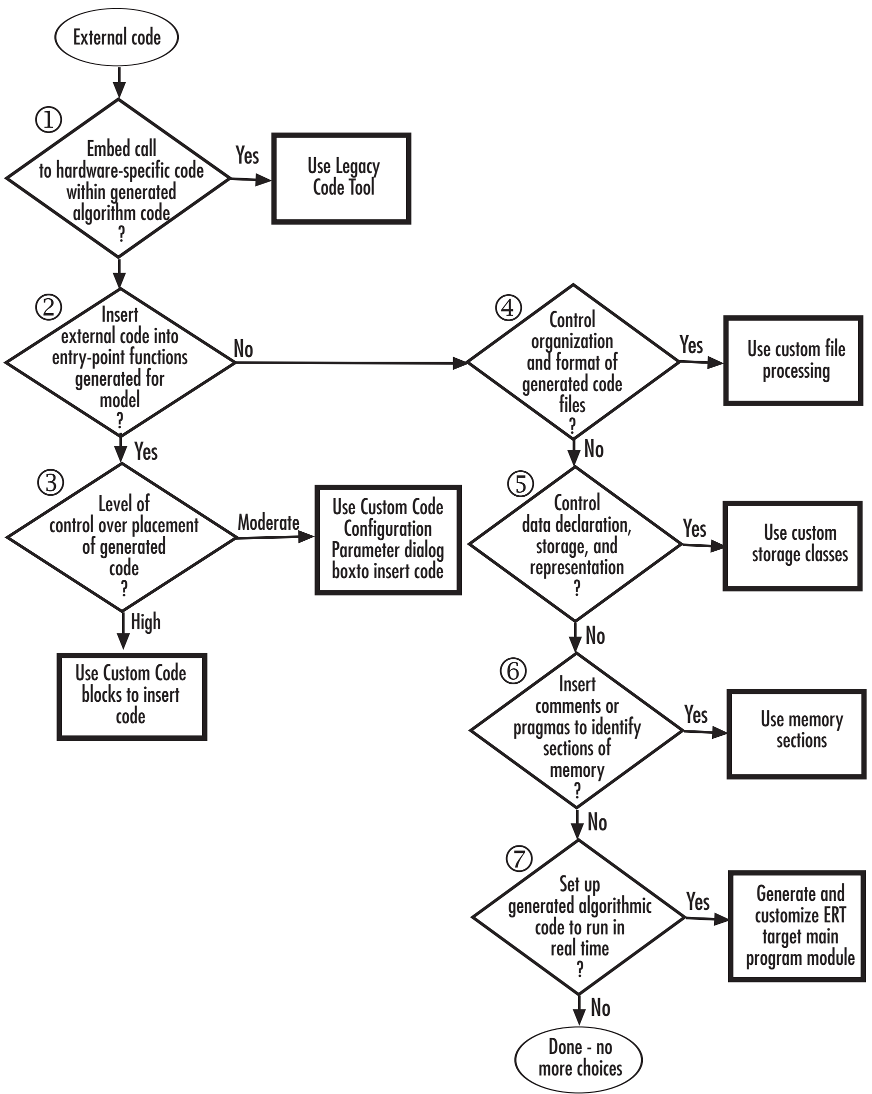
If...	Consider...	For More Information, See...
Generated code must use math functions and operators that are consistent with imported external code	*EC only*—Using the Code Replacement Tool, code replacement library (CRL) API, and Code Replacement Viewer to create, examine, validate, and register function and operator code replacement tables	• `rtwdemo_crl_script` • "What Is Code Replacement?" and "What Is Code Replacement Customization?" in the Embedded Coder documentation

If...	Consider...	For More Information, See...
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, and data type replacement to match data type, formatting, and storage that is consistent with that used by the imported external code	• "Data Types" • "Data Representation"
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	• `rtwdemo_symbols` • `rtwdemo_namerules` • "Construction of Generated Identifiers" 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	• `rtwdemo_comments` • "Configure Code Comments" 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	*EC only*—In the Configuration Parameters dialog box, on the **Code Style** pane, configure the style for the generated code	• `rtwdemo_parentheses` • "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	• "Integrate Device Drivers" • "Integrate C Functions Using Legacy Code Tool" in the Simulink documentation • "Integrate External Code Using Legacy Code Tool"
2	Insert target-specific C or C++ code into entry-point functions that Simulink Coder generates for a model with a high level of control over code placement; for example, inserting startup, initialization, or termination code	Use Custom Code blocks	• `rtwdemo_slcustcode` • "Insert Custom Code Blocks"
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	• Integrating the Generated Code into the External Environment • "Configure Model for External Code Integration"
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	*EC only*—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	• `rtwdemo_codetemplate` • "Customize Code Organization and Format" in the Embedded Coder documentation

	If You Need To...	Then...	For More Information, See...
5	Control how the Simulink Coder product declares, stores, and represents signals, tunable parameters, block states, and data objects in generated code	*EC only*— Design (create) and apply custom storage classes	• `rtwdemo_cscpredef` • `rtwdemo_importstruct` • `rtwdemo_advsc` • “Custom Storage Classes” in the Embedded Coder documentation
6	Insert comments or `pragmas` in generated code to identify memory for custom storage classes or model- or subsystem-level functions and internal data	*EC only*— Use the memory section capability	• `rtwdemo_memsec` • “About Memory Sections”
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)	*EC only*—Generate and customize an ERT target main (harness) program module (`ert_main.c` or `ert_main.cpp`) for the model	• “Deployment” • “Standalone Programs (No Operating System)”

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 Simulink Coder generates for interfacing with the external code	Use Custom Code blocks	• `rtwdemo_slcustcode` • "Insert Custom Code Blocks"
Pass composite data	Represent the data in the model as a vector or bus	• `rtwdemo_scalarrep` • `rtwdemo_slbus` • "Composite Signals" • "Optimize Code Generated for Vector Assignments" and"Buses"
Read from or write to a specific region or area of memory	*EC only*— 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) `GetSet`	• "Increase Code Efficiency With GetSet CSC" • "GetSet Custom Storage Class" in the Embedded Coder documentation
Generate a C++ class interface to the model code — encapsulating model data as properties	Select `C++ class` code interface packaging in the Configuration Parameters dialog box or programmatically with	• "Simple Use of C++ Class Control" • "C++ Class Interface Control" in the Embedded Coder documentation

If You Need To...	Then...	For More Information, See...
and model entry-point functions as methods	MATLAB commands — customizing the generated C++ class interface is *EC only*	
Control how Embedded Coder generates function prototypes — arguments, argument order, and data types—for a model (for example, so the prototypes match the external code)	*EC only*— 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	• "Sample Procedure for Configuring Function Prototypes" • "Function Prototype Control" 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	• Integrating the Generated Code into the External Environment • "Configure Model for External Code Integration"

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	*EC only*— 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	• "Techniques for Exporting Function-Call Subsystems" • `rtwdemo_exporting_functions` • "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 (`shrlib.tlc`) to build a Windows dynamic link library (`.dll`) or a UNIX shared object (`.so`) file from a model. An application, which runs on a Windows or a UNIX 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 "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
  ```

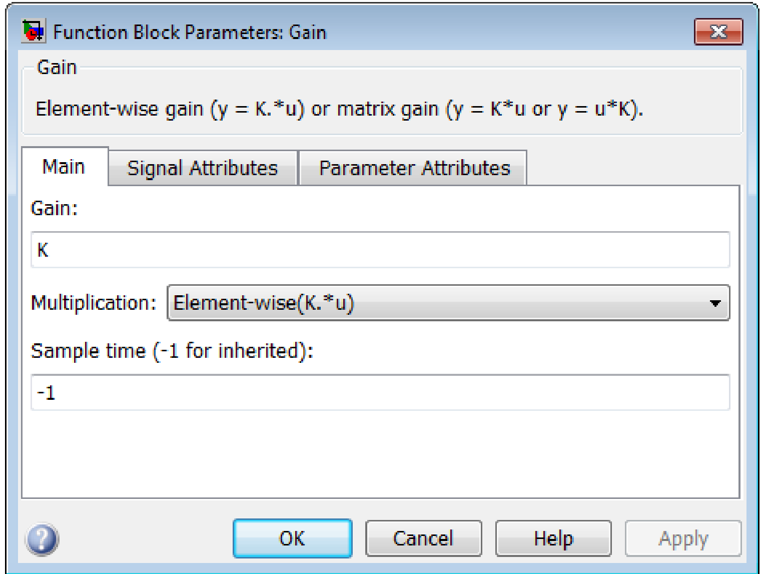
  The prototype serves as a function wrapper. If your compiler supports C++ code, the value `__cplusplus` is defined. The linkage specification `extern "C"` specifies C linkage with no name mangling.
- If the generated code is in C and your legacy or custom code is in C++, include an `extern "C"` linkage 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

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 14-34. 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.

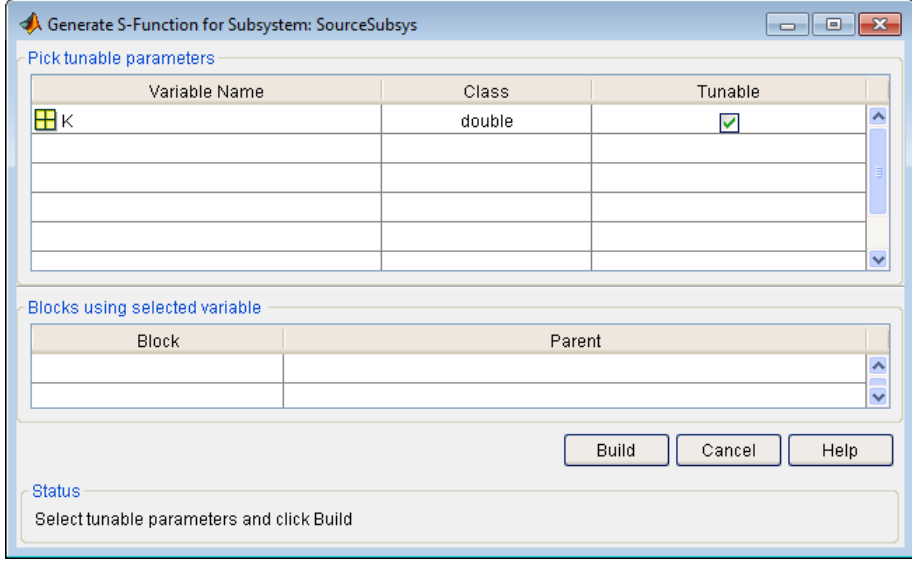
   Alternatively, you can right-click the subsystem and select **C/C++ Code** > **Generate S-Function** from the subsystem block's context menu.

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.

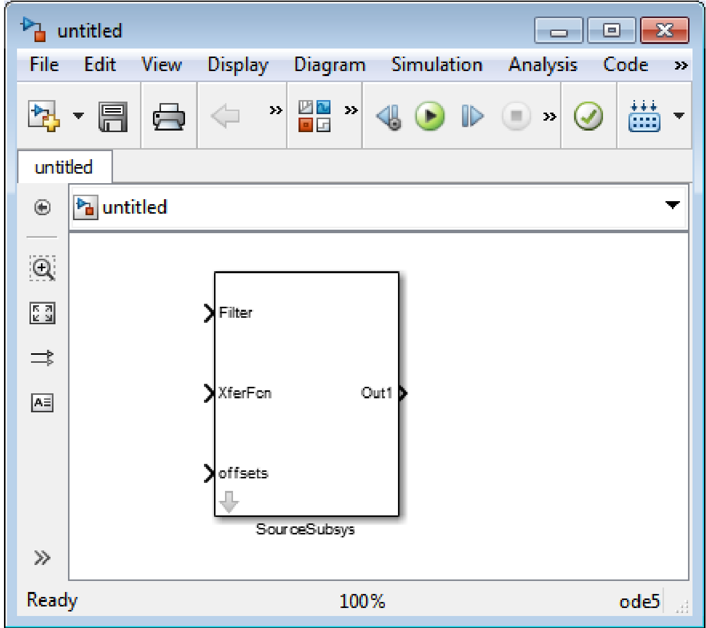
When you select a parameter in the upper pane, the lower pane shows the blocks that reference the parameter, and the parent system of each such block.



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.mexw32`)

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 14-32.

---

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))));
```

Notes

- In automatic S-function generation, the **Use Value for Tunable Parameters** option is cleared or at the command line set to `‘off’`.
- A MEX S-function wrapper must only be used in the MATLAB version in which the wrapper is created.

Integrate External Code Using Legacy Code Tool

In this section...

"Legacy Code Tool and Code Generation" on page 16-25
"Generate Inlined S-Function Files for Code Generation" on page 16-26
"Apply Code Style Settings to Legacy Functions" on page 16-26
"Address Dependencies on Files in Different Locations" on page 16-27
"Deploy S-Functions for Simulation and Code Generation" on page 16-28

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 Simulink Coder build process looks for the generated `rtwmakecfg.m` file in the same

folder as the S-function's 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 Simulink Coder 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”.

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.

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 *model*.c or *model*.cpp file, outside of any function	**Source file** and enter the custom code to insert. **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 #include near the top of the *model*.c or *model*.cpp 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.
Insert custom code near the top of the generated *model*.h file	**Header file** and enter the custom code to insert.
Insert custom code inside the model's initialize function in the *model*.c or *model*.cpp file	**Initialize function**
Insert custom code inside the model's terminate function in the *model*.c or *model*.cpp 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...

	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 `inc`, the build process adds a rule similar to the following:
	
	`%.obj: buildir\inc\%.c`
	`        $(CC) -c -Fo$(@F) $(CFLAGS) $<`
	
	The build process adds the rules in the order you list the source files.

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.
Use the same custom code settings as those specified for simulation of MATLAB Function blocks, Stateflow charts, and Truth Table blocks	**Use the same custom code settings as Simulation Target** **Note:** This option refers to the **Simulation Target** pane in the Configuration Parameters dialog box.
Enable a library model to use custom code settings unique from the parent model to which the library is linked	**Use local custom code settings (do not inherit from main model)** **Note:** This option is available only for library models that contain MATLAB Function blocks, Stateflow charts, or Truth Table blocks.

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 “Code Generation Pane: Custom Code”.

Insert Custom Code Blocks

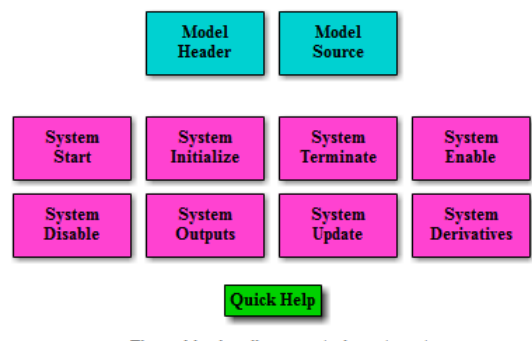
The following sections explain how to use blocks in the Custom Code block library to insert custom code into the code generated for a model. This chapter includes the following topics:

In this section...
“Custom Code Library” on page 16-32
“Embed Custom Code Directly Into MdlStart Function” on page 16-35
“Custom Code in Subsystems” on page 16-38
“Preserve User Files in Build Folder” on page 16-39

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 generated by the Simulink Coder product 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 library. You can access the Simulink Coder library by using the Simulink Library Browser. You can access Custom Code blocks by using the Simulink Coder 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`.



Note: If you need to integrate custom C++ code with generated C code or vice versa, see "Modify External Code for Language Compatibility".

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. However, when simulation targets for referenced models are generated, Custom Code blocks within them are ignored. On the other hand, when referenced model code is generated to create Simulink Coder targets, custom code is included and is compiled in the generated 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 library) in the Simulink Library Browser
- Right-clicking the Custom Code sublibrary icon in the right pane of the Simulink Library Browser

The latter method opens the window shown in the previous section.

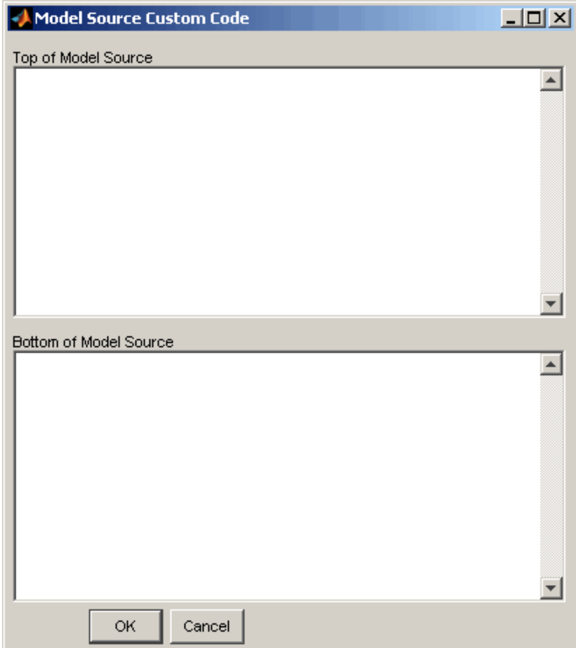
The two blocks on the top row contain text fields for inserting custom code at the top and bottom of

- `*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.



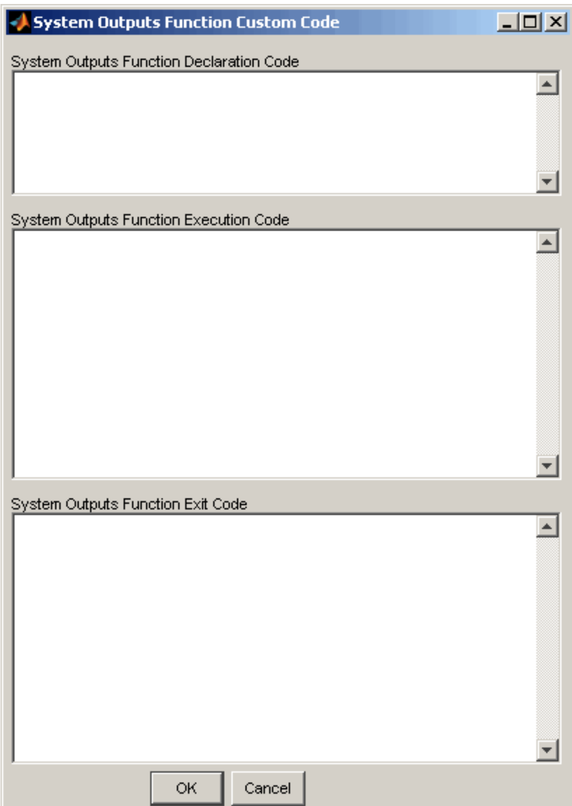
The eight function blocks in the second and third rows contain text fields to insert custom code sections at the top and bottom of these designated model functions:

- 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:

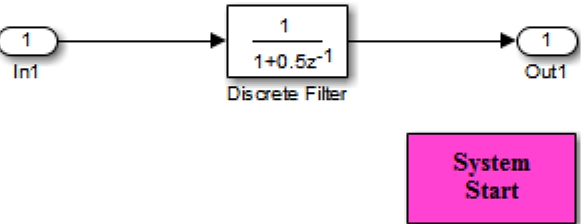
- Declaration code

- Execution code
- Exit code

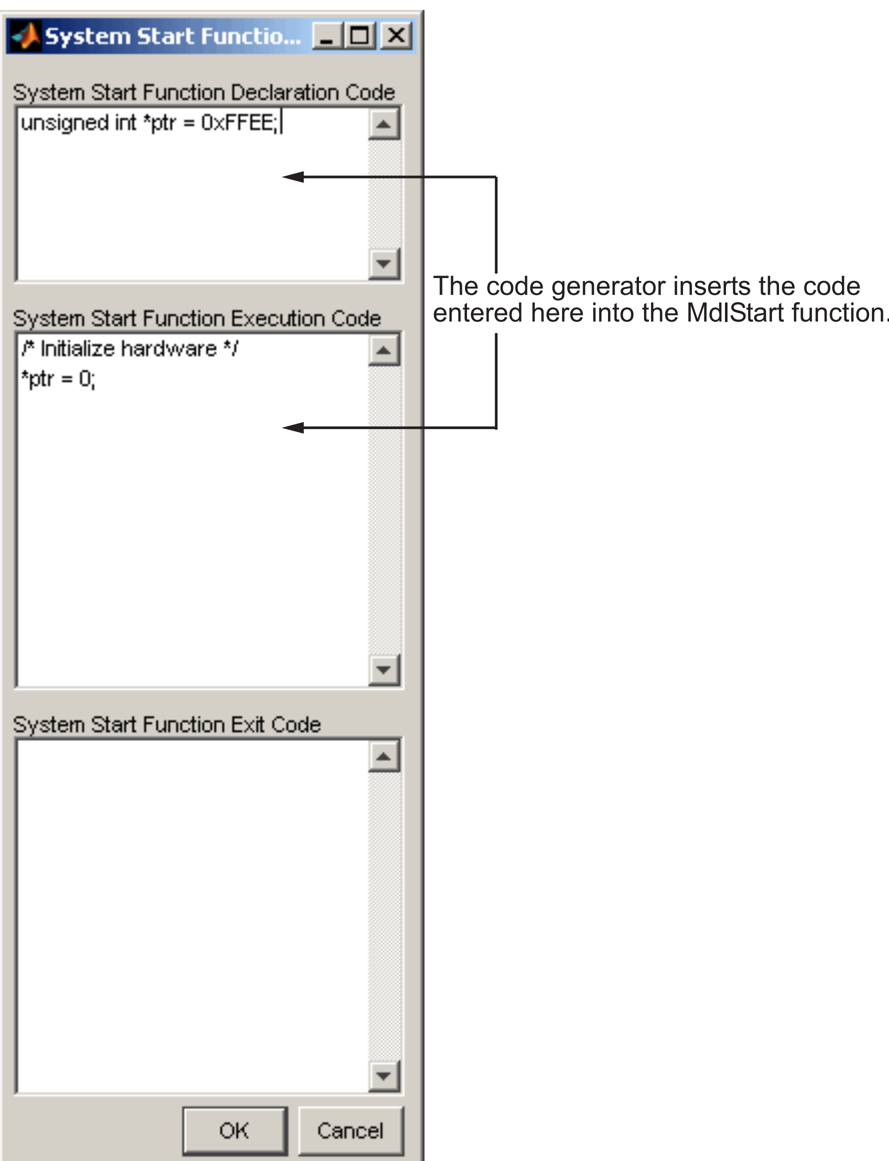


Embed Custom Code Directly Into MdlStart Function

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 Simulink Coder product from deleting it during the TLC code generation process, insert the string `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 string "target specific file" as shown in the preceding example, and that the string is in the first line of the source file. Other text can appear before or after this string.

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 string). The indenting is harmless, but can cause differences to be detected by source control software that might trigger unnecessary updates.

Insert S-Function Code

In this section...

"About S-Functions and Code Generation" on page 16-40
"Write Noninlined S-Functions" on page 16-45
"Write Wrapper S-Functions" on page 16-47
"Write Fully Inlined S-Functions" on page 16-56
"Write Fully Inlined S-Functions with mdlRTW Routine" on page 16-56
"Guidelines for Writing Inlined S-Functions" on page 16-75
"Write S-Functions That Support Expression Folding" on page 16-76
"S-Functions That Specify Port Scope and Reusability" on page 16-87
"S-Functions That Specify Sample Time Inheritance Rules" on page 16-91
"S-Functions That Support Code Reuse" on page 16-93
"S-Functions for Multirate Multitasking Environments" on page 16-94
"Build Support for S-Functions" on page 16-100

About S-Functions and Code Generation

This section describes how to create S-functions that work seamlessly with Simulink Coder code generation. It begins with basic concepts and concludes with an example of how to create a highly optimized direct-index lookup table S-Function block.

This section assumes that you understand the following concepts:

- Level 2 S-functions
- Target Language Compiler (TLC) scripting
- How the Simulink Coder software generates and builds C/C++ code

---

Note When this section refers to actions performed by the Target Language Compiler, including parsing, caching, creating buffers, and so on, the name Target Language Compiler is spelled out fully. When referring to code written in the Target Language Compiler syntax, this section uses the abbreviation TLC.

Note The guidelines presented in this section are for Simulink Coder users. Even if you do not currently use the Simulink Coder code generator, you should follow the practices presented in this section when writing S-functions, especially if you are creating general-purpose S-functions.

Classes of Problems Solved by S-Functions

S-functions help solve various kinds of problems you might face when working with the Simulink and Simulink Coder products. These problems include

- Extending the set of algorithms (blocks) provided by the Simulink and Simulink Coder products
- Interfacing legacy (hand-written) code with the Simulink and Simulink Coder 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 Simulink Coder 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 section, you can create S-functions for most applications that work with the Simulink Coder generated code.

Although S-functions provide a generic and flexible solution for implementing complex algorithms in a Simulink 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 Simulink Coder product to inline your S-functions. If you are producing an S-function for existing code, consider using the Simulink Legacy Code Tool.

Types of S-Functions

The implementation of S-functions changes based on your requirements. This section discusses the typical problems that you may face and how to create S-functions for

applications that need to work with the Simulink and Simulink Coder 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 Simulink Coder products automatically.”
2. “I have a lot of hand-written code that I need to interface. I want to call my function from the Simulink and Simulink Coder products in an efficient manner.”

   or said another way:

   “I want to create a block for my blockset that will be distributed throughout my organization. I'd like it to be very maintainable with efficient code. I'd like my algorithm to exist in one place but work with both the Simulink and Simulink Coder products.”
3. “I want to implement a highly optimized algorithm in the Simulink and Simulink Coder products that looks like a built-in block and generates very efficient code.”

MathWorks products have adopted terminology for these different requirements. Respectively, the situations described above map to this terminology:

1. Noninlined S-function
2. Wrapper S-function
3. Fully inlined S-function

Noninlined S-Functions

A noninlined S-function is a C or C++ MEX S-function that is treated identically by the Simulink engine and Simulink Coder 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 Simulink Coder 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 Simulink Coder build process in the sense that once you build a MEX S-function in your model, there is no additional preparation prior to clicking

Build in the **Code Generation** pane of the Configuration Parameters dialog box for your model.

Some restrictions exist concerning the names and locations of noninlined S-function files when generating makefiles. See “Write Noninlined S-Functions” on page 16-45.

Wrapper S-Functions

A wrapper S-function is ideal for interfacing hand-written code or a large algorithm that is encapsulated within a few procedures. In this situation, usually the procedures reside in modules that are separate from the MEX S-function. The S-function module typically contains a few calls to your procedures. Because the S-function module does not contain any parts of your algorithm, but only calls your code, it is referred to as a *wrapper S-function*.

In addition to the MEX S-function wrapper, you need to create a TLC wrapper that complements your S-function. The TLC wrapper is similar to the S-function wrapper in that it contains calls to your algorithm.

Fully Inlined S-Functions

For S-functions to work in the Simulink environment, some overhead code is generated. When the Simulink Coder software generates 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. In Simulink Coder terminology, inline means to specify a text string 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 Simulink Coder 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 Simulink Coder 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.

The Simulink 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 “Integrate External Code Using Legacy Code Tool” on page 16-25.

Basic Files Required for Implementation

This section briefly describes what files and functions you need to create noninlined, wrapper, and fully inlined S-functions.

- Noninlined S-functions require the C or C++ MEX S-function source code (`sfunction.c` or `sfunction.cpp`).
- Wrapper S-functions that inline a call to your algorithm (your C/C++ function) require an `sfunction.tlc` file.
- Fully inlined S-functions also 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 at the end of this section contains two operating modes — one for evenly spaced `x-data` and one for unevenly spaced `x-data`.

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 Simulink Coder 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

- Use only C MEX S-functions with the Simulink Coder code generator. You cannot use Level-1 MATLAB language S-functions with Simulink Coder software.
- To inline an S-function, use the Legacy Code Tool. The Legacy Code Tool automatically generates fully inlined C MEX S-functions for legacy or custom code. In addition, the tool generates other files for compiling and building the S-function for simulation and generate a masked S-function block configured to call existing external code. For more information, see "Integrate C Functions Using Legacy Code Tool" in the Simulink documentation and "Integrate External Code Using Legacy Code Tool".
- If you are rapid prototyping, you might not have to inline an S-function.. If you choose not to inline the C MEX S-function, write the S-function, include it directly in the model, and let the Simulink Coder software generate the code. For more information, see "Write Noninlined S-Functions" on page 16-45.

Write Noninlined S-Functions

- "About Noninlined S-Functions" on page 16-45
- "Guidelines for Writing Noninlined S-Functions" on page 16-45
- "Noninlined S-Function Parameter Type Limitations" on page 16-47

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 32–bit Microsoft Windows system, the file name would be `sfunction.mexw32`. 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 string.
- 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 Simulink Coder build process with the names of the modules used to build the S-function. You can do this by using the Simulink Coder 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, you must force the coder to rebuild the top model, as explained in "Control Regeneration of Top Model Code".

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 file *`matlabroot`*`/rtw/c/src/rt_matrx.h(.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-Functions

- "About Wrapper S-Functions" on page 16-47
- "MEX S-Function Wrapper" on page 16-48
- "TLC S-Function Wrapper" on page 16-52
- "The Inlined Code" on page 16-55

About Wrapper S-Functions

This section describes how to create S-functions that work seamlessly with the Simulink and Simulink Coder products using the *wrapper* concept. This section 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

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.

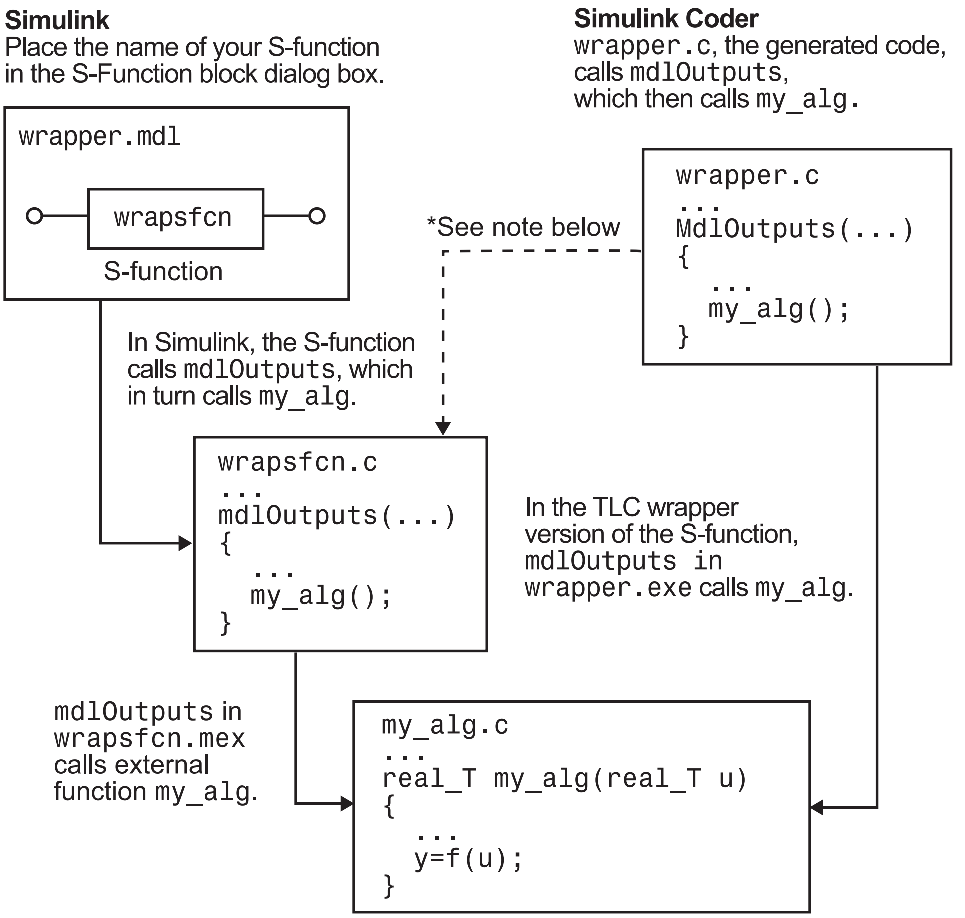
You can integrate `my_alg` into generated code (that is, embed the call to `my_alg` in the generated code) by creating a TLC S-function wrapper (for example, `wrapsfcn.tlc`). The advantage of creating a TLC S-function wrapper is that the empty function calls can be eliminated and the overhead of executing 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)

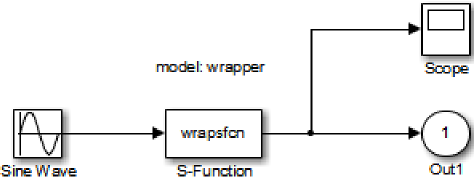
then you must create a *fully inlined TLC file* for your S-function.

The next figure shows the wrapper S-function concept.



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);
}
```

See the section “Header Dependencies When Interfacing Legacy/Custom Code with Generated Code” on page 11-39 in the Simulink Coder documentation for more information.

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 section 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 Simulink Coder software generates 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

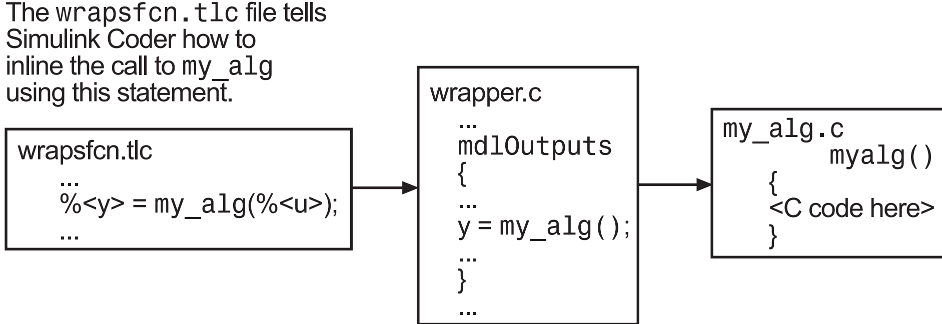
The generated code makes the call to your S-function, `wrapsfcn.c`, in `mdlOutputs` by using this code:

```
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.



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 */

  /* Outport 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

Continuing the example of the previous section, 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 */

  /* Outport 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

- “About S-Functions and mdlRTW” on page 16-57
- “S-Function RTWdata” on page 16-57

- "Direct-Index Lookup Table Algorithm" on page 16-58
- "Direct-Index Lookup Table Example" on page 16-59

About S-Functions and mdlRTW

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 section 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 section 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 Simulink Coder generated code 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

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 strings 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(gcb,'RTWdata',mydata)
get_param(gcb,'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.

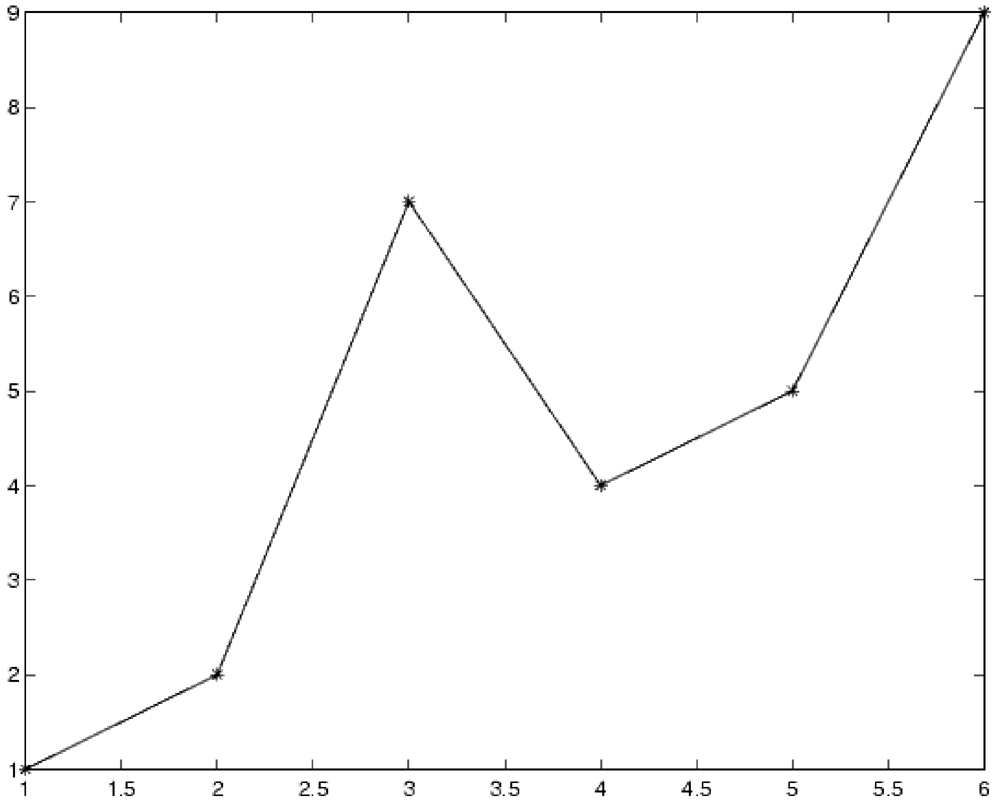
This direct 1-D lookup example computes an approximate solution *p(x)* to a partially known function *f(x)* at *x=x0*, given data point pairs *(x,y)* in the form of an *x*-data vector and a *y*-data vector. For a given data pair (for example, the *i'th* pair), *y_i = f(x_i)*. It is assumed that the *x*-data values are monotonically increasing. If *x0* 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 section 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 16-61).

Error Handling

In this example, the `mdlCheckParameters` routine verifies that

- The new parameter settings 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.

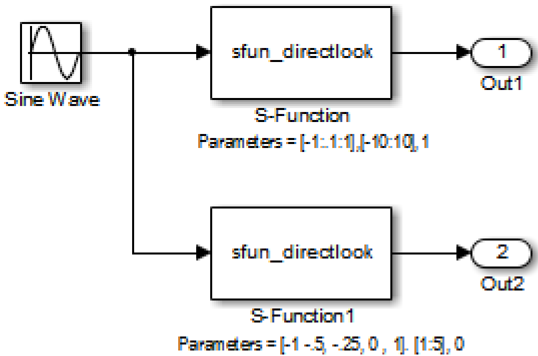
mdlRTW Usage

The Simulink Coder 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 Simulink Coder build process to use the module `lookup_index.c` when creating the executable.

When generating code for this model, the Simulink Coder software uses the S-function's `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 output (root outports fed by signals with auto storage) */
ExternalOutputs_sfun_directlook_ex sfun_directlook_ex_Y;

/* Real-time model */
rtModel_sfun_directlook_ex sfun_directlook_ex_M_;
rtModel_sfun_directlook_ex *sfun_directlook_ex_M = &sfun_directlook_ex_M_;

/* Model output function */
static void sfun_directlook_ex_output(int_T tid)
{

  /* local block i/o variables */

  real_T rtb_SFunction_h;
  real_T rtb_temp1;

  /* Sin: '<Root>/Sine Wave' */
  rtb_temp1 = sfun_directlook_ex_P.SineWave_Amp *
    sin(sfun_directlook_ex_P.SineWave_Freq * sfun_directlook_ex_M->Timing.t[0] +
    sfun_directlook_ex_P.SineWave_Phase) + sfun_directlook_ex_P.SineWave_Bias;

  /* Code that is inlined for the top S-function block in the
   * sfun_directlook_ex model
   */
```

```
  /* S-Function Block: <Root>/S-Function */
  {
    const real_T *xData = &sfun_directlook_ex_P.SFunction_XData[0];
    const real_T *yData = &sfun_directlook_ex_P.SFunction_YData[0];
    real_T spacing = xData[1] - xData[0];

    if ( rtb_temp1 <= xData[0] ) {
      rtb_SFunction_h = yData[0];
    } else if ( rtb_temp1 >= yData[20] ) {
      rtb_SFunction_h = yData[20];
    } else {
      int_T idx = (int_T)( ( rtb_temp1 - xData[0] ) / spacing );
      rtb_SFunction_h = yData[idx];
    }
  }

  /* Outport: '<Root>/Out1' */
  sfun_directlook_ex_Y.Out1 = rtb_SFunction_h;

/* Code that is inlined for the bottom S-function block in the
   * sfun_directlook_ex model
   */
  /* S-Function Block: <Root>/S-Function1 */
  {
    const real_T *xData = &sfun_directlook_ex_P.SFunction1_XData[0];
    const real_T *yData = &sfun_directlook_ex_P.SFunction1_YData[0];
    int_T idx;

    idx = GetDirectLookupIndex(xData, 5, rtb_temp1);
    rtb_temp1 = yData[idx];
  }

  /* Outport: '<Root>/Out2' */
  sfun_directlook_ex_Y.Out2 = rtb_temp1;
}

/* Model update function */
static void sfun_directlook_ex_update(int_T tid)
{

  /* Update absolute time for base rate */

  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 * 0x10000;

  {
    /* Update absolute timer for sample time: [0.1s, 0.0s] */

    if(!(++sfun_directlook_ex_M->Timing.clockTick1))
    ++sfun_directlook_ex_M->Timing.clockTickH1;
    sfun_directlook_ex_M->Timing.t[1] = sfun_directlook_ex_M->Timing.clockTick1
```

```
      * sfun_directlook_ex_M->Timing.stepSize1 +
      sfun_directlook_ex_M->Timing.clockTickH1 *
      sfun_directlook_ex_M->Timing.stepSize1 * 0x10000;
  }
}
...
```

***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 1990-2004 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() */
#if defined(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.
     */
    {
        int_T     i;
        boolean_T spacingEqual;
        real_T    *xData = mxGetPr(XVECT(S));
```

```
        int_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.
     */
#if defined(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);
    int_T             ny     = ssGetOutputPortWidth(S,0);
    int_T             xLen   = mxGetNumberOfElements(XVECT(S));
    int_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 1990-2004 The MathWorks, Inc.
 */
#include "tmwtypes.h"

/*
 * 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 1990-2004 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”). This property is a structure of strings 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 Simulink Coder generates for the block, will include 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

Write S-Functions That Support Expression Folding

- "About S-Functions that Support Expression Folding" on page 16-76
- "Categories of Output Expressions" on page 16-77
- "Acceptance or Denial of Requests for Input Expressions" on page 16-82
- "Expression Folding in a TLC Block Implementation" on page 16-83

About S-Functions that Support Expression Folding

This section 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 section 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 16-77).

The sections 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 outport
- 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.

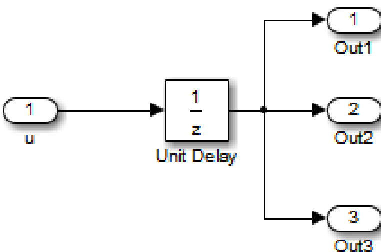
A *trivial* output expression is an expression that can be repeated, without a performance penalty, when the output port has multiple output destinations. For example, the output of a Unit Delay block is defined as a trivial expression because the output expression is simply a direct access to the block's 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.

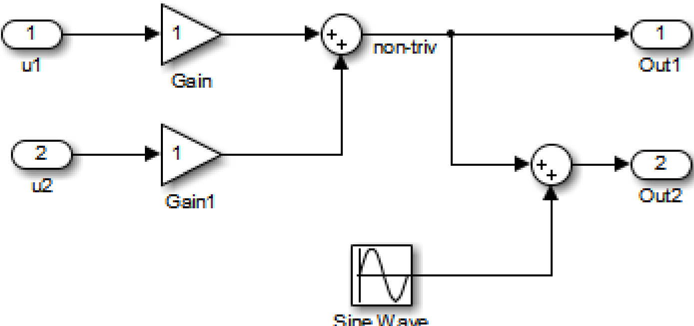


The following code excerpt shows code generated from the Unit Delay block in this block diagram. The three root outputs are directly assigned from the state of the Unit Delay block, which is stored in a field of the global data structure `rtDWork`. Since the assignment is direct, without expressions, there is no performance penalty associated with using the trivial expression for multiple destinations.

```
void MdlOutputs(int_T tid)
{
   ...
  /* Outport: <Root>/Out1 incorporates:

   *   UnitDelay: <Root>/Unit Delay */
  rtY.Out1 = rtDWork.Unit_Delay_DSTATE;

  /* Outport: <Root>/Out2 incorporates:
   *   UnitDelay: <Root>/Unit Delay */
  rtY.Out2 = rtDWork.Unit_Delay_DSTATE;

  /* Outport: <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.

Category of Expression	When to Use
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 section 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 16-83.

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 16-82.
- Output ports, as explained in “Categories of Output Expressions” on page 16-77.

This section 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/utillib.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 have overridden one of the TLC block implementations provided by the Simulink Coder product 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 sections.

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,lcv,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

- `lcv`: 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`: string 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 text string for the output signal or address. The string 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,lcv,idx,retType) void
  %switch retType
    %case "Signal"
      %return LibBlockParameter(Value,ucv,lcv,idx)
    %case "SignalAddr"
      %return LibBlockParameterAddr(Value,ucv,lcv,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,lcv,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, lcv, idx)
%assign u1 = LibBlockInputSignal(1, ucv, lcv, 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,lcv,idx,retType) void
%switch retType
%case "Signal"
   %assign u = LibBlockInputSignal(portIdx, ucv, lcv, 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, lcv = RollThreshold, block, "Roller", rollVars
%assign u0 = LibBlockInputSignal(0, "", lcv, sigIdx)
 %assign u1 = LibBlockInputSignal(1, "", lcv, 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 string 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 string, 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 Simulink Coder software can declare individual local variables for the input and output ports
- `SS_REUSABLE_AND_LOCAL`: Indicates that the Simulink Coder software 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

---

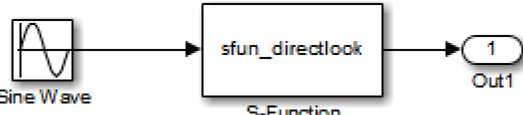
Note Marking an input or output port as a local variable does not imply that the code generator uses a local variable in the generated code. If your S-function accesses the inputs and outputs only in its `mdlOutputs` routine, the code generator declares the inputs and outputs as local variables. However, 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, overwritable
 	
Outputs: Local, reusable
	`ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);`
	
	`ssSetInputPortOverWritable(S, 0, TRUE);`
	
	`ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_LOCAL);`
		The *`model`*`.c` file declares a local variable in the output function.
		
		`/* local block i/o variables */`
		`real_T rtb_SFunction[2];`

Inputs: Global, reusable, overwritable
 	
Outputs: Global, reusable
	`ssSetInputPortOptimOpts(S, 0, SS_REUSABLE_AND_GLOBAL);`
	
	`ssSetInputPortOverWritable(S, 0, TRUE);`
	
	`ssSetOutputPortOptimOpts(S, 0, SS_REUSABLE_AND_GLOBAL);`
		The *`model`*`.h` file defines a block signals structure with a single element to store the S-function's input and output.
		
		`/* Block signals (auto storage) */`
		`typedef struct {`
		`real_T SFunction[2];`
		`} BlockIO_sl_directlook;`

Scope and reusability	S-function `mdlInitializeSizes` code	Generated code

		The *`model`*`.c` file uses this element of the structure in calculations of the S-function's input and output signals.
		` /* Sin: '<Root>/Sine Wave' */`
		`sl_directlook_B.SFunction[0] = sin ...`
		`/* snip */`
		`/*S-Function Block:<Root>/S-Function*/`
		`{`
		`const real_T *xData =`
		`  &sl_directlook_P.SFunction_XData[0]`

Inputs: Local, not reusable
 	
Outputs: Local, not reusable
	`ssSetInputPortOptimOpts(S, 0,`
	`SS_NOT_REUSABLE_AND_LOCAL);`
	
	`ssSetInputPortOverWritable(S, 0,`
	`FALSE);`
	
	`ssSetOutputPortOptimOpts(S, 0,`
	`SS_NOT_REUSABLE_AND_LOCAL);`
		The *`model`*`.c` file declares local variables for the S-function's input and output in the output function
		`/* local block i/o variables */`
		`  real_T rtb_SineWave[2];`
		`  real_T rtb_SFunction[2];`

Inputs: Global, not reusable
 	
Outputs: Global, not reusable
	`ssSetInputPortOptimOpts(S, 0,`
	`SS_NOT_REUSABLE_AND_GLOBAL);`
	
	`ssSetInputPortOverWritable(S, 0,`
	`FALSE);`
	
	`ssSetOutputPortOptimOpts(S, 0,`
	`SS_NOT_REUSABLE_AND_GLOBAL);`
		The *`model`*`.h` file defines a block signal structure with individual elements to store the S-function's input and output.
		`/* Block signals (auto storage) */`
		`typedef struct {`
		`  real_T SineWave[2];`
		`  real_T SFunction[2];`
		`} BlockIO_sl_directlook;`
		The *`model`*`.c` file uses the different elements in this structure when calculating the S-function's input and output.
		` /* 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]`

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.	`ssSetModelReferenceSampleTimeInheritanceRule (S, USE_DEFAULT_FOR_DISCRETE_INHERITANCE)`

If...	Use...
Any of the macros or functions are used for • Throwing errors if sample time is inherited, continuous, or constant • Checking `ssIsSampleHit` • Checking whether sample time is inherited in either `mdlSetInputPortSampleTime` or `mdlSetOutputPortSampleTime` before setting	`ssSetModelReferenceSampleTimeInheritanceRule...` `(S,USE_DEFAULT_FOR_DISCRETE_INHERITANCE)`
The S-function uses its sample time for computing parameters, outputs, and so on	`ssSetModelReferenceSampleTimeInheritanceRule` `(S, DISALLOW_SAMPLE_TIME_INHERITANCE)`

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 **Diagnostics/Solver** 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”. 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 16-94
- "Rate Grouping Support in S-Functions" on page 16-94
- "Create Multitasking, Multirate, Port-Based Sample Time S-Functions" on page 16-95

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 Embedded Coder product generates 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 Simulink Coder software 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 outport 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(tslow)`
- Input: local and reusable
- Output: local and reusable

- DirectFeedthrough: yes

```
OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSepcialSampleHit("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 = 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
            outputing. 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(tslow-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(tfast) + u(tslow-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;}
```

Build Support for S-Functions

- “About Build Support for S-Functions” on page 16-101
- “Implicit Build Support” on page 16-101
- “Specify Additional Source Files for an S-Function” on page 16-102
- “Use TLC Library Functions” on page 16-103
- “Use rtwmakecfg.m API to Customize Generated Makefiles” on page 16-103
- “Precompile S-Function Libraries” on page 16-108

About Build Support for S-Functions

User-written S-Function blocks provide a powerful way to incorporate legacy and custom code into the Simulink and Simulink Coder development environment. In most cases, you should use S-functions to integrate existing code with Simulink Coder generated code. Several approaches to writing S-functions are available as discussed in

- "Write Noninlined S-Functions" on page 16-45
- "Write Wrapper S-Functions" on page 16-47
- "Write Fully Inlined S-Functions" on page 16-56
- "Write Fully Inlined S-Functions with mdlRTW Routine" on page 16-56
- "S-Functions That Support Code Reuse" on page 16-93
- "S-Functions for Multirate Multitasking Environments" on page 16-94

S-functions also provide the most flexible and capable way of including build information for legacy and custom code files in the Simulink Coder build process.

This section discusses the different ways of adding S-functions to the Simulink Coder build process.

Implicit Build Support

When building models with S-functions, the code generator automatically adds rules, include paths, and source filenames to the generated makefile. For this to occur, the source files (`.h`, `.c`, and `.cpp`) for the S-function must be in the same folder as the S-function MEX-file. The code generator propagates this information through the token expansion mechanism of converting a template makefile (TMF) to a makefile. The propagation requires the TMF to support the tokens.

Details of the implicit build support follow:

- 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 Simulink Coder code generator adds a makefile rule for compiling files from that folder.
- When an S-function is not inlined with a TLC file, the Simulink Coder code generator must compile the S-function's source file. To determine the name of the source file to add to the list of files to compile, the code generator searches for `*sfcnname*.cpp`

on the MATLAB path. If the source file is found, the code generator adds the source filename to the makefile. If `sfcnname.cpp` is not found on the path, the code generator adds the filename `sfcnname.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. You can do this by specifying 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, as in

```
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:

- Specifying `sfun_main`, `sfun_module1`, and `sfun_module2` in the **S-function modules** field in the S-Function block dialog box
- Entering the following command at the MATLAB command prompt:

  ```
  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 Simulink Coder software must be able to find the additional source files when executing the makefile. For the Simulink Coder software to locate the additional files, place them in the same folder as the S-function MEX-file. This will enable you to leverage the implicit build support discussed in “Implicit Build Support” on page 16-101.

When you are ready to generate code after using the **S-function modules** field or `SFunctionModules` parameter, you must force the coder to rebuild the top model, as explained in "Control Regeneration of Top Model Code".

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 explained in "Use rtwmakecfg.m API to Customize Generated Makefiles" on page 16-103.

Use TLC Library Functions

If you inline your S-function by writing a TLC file, you can add source filenames 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 the Simulink Coder software 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 explained in "Use rtwmakecfg.m API to Customize Generated Makefiles" on page 16-103.

Use rtwmakecfg.m API to Customize Generated Makefiles

- "Overview" on page 16-103
- "Create the rtwmakecfg Function" on page 16-104
- "Modify the Template Makefile" on page 16-107

Overview

Simulink Coder 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` 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 MATLAB language function `rtwmakecfg` in a file `rtwmakecfg.m`. The Simulink Coder software associates this file with your S-function based on its folder location. “Create the rtwmakecfg Function” on page 16-104 discusses the requirements for the `rtwmakecfg` function and the data it should return.
2. Modify your target's TMF such that it supports macro expansion for the information returned by `rtwmakecfg` functions. “Modify the Template Makefile” on page 16-107 discusses the required modifications.

After the TLC phase of the build process, when generating a makefile from the TMF, the Simulink Coder code generator searches for an `rtwmakecfg.m` file in the folder that contains the S-function component. If it finds the file, the code generator 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 (*`sfcname.mexext`* on a Microsoft Windows system and *`sfcname`* and a platform-specific extension on The Open Group UNIX system). The function must return a structured array that contains the following fields:

Field	Description
`makeInfo.includePath`	A cell array that specifies additional include folder names, organized as a row vector. The Simulink Coder code generator expands the folder names into include instructions in the generated makefile.
`makeInfo.sourcePath`	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

Field	Description
	SFunctionModules parameter if they are not in the same folder as the S-function. The Simulink Coder code generator expands the folder names into make rules in the generated makefile.
makeInfo.sources	A cell array that specifies additional source filenames (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 SFunctionModules parameter. The Simulink Coder code generator expands the filenames into make variables that contain the source files. You should specify only filenames (with extension). Specify path information with the sourcePath field.
makeInfo.linkLibsObjs	A cell array that specifies additional, fully qualified paths to object or library files against which the Simulink Coder generated code should link. The Simulink Coder code generator does not compile the specified objects and libraries. However, it includes them when linking the final executable. This can be useful for incorporating libraries that you do not want the Simulink Coder code generator to recompile or for which the source files are not available. You might 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 Simulink Coder build process.
makeInfo.precompile	A Boolean flag that indicates whether the libraries specified in the rtwmakecfg.m file exist in a specified location (precompile==1) or if the libraries need to be created in the build folder during the Simulink Coder build process (precompile==0).
makeInfo.library	A structure array that specifies additional run-time libraries and module objects, organized as a row vector. The Simulink Coder code generator expands the information into make rules in the generated makefile. See the next table for a list of the library fields.

The makeInfo.library field consists of the following elements:

Element	Description
makeInfo.library(n).Name	A character array that specifies the name of the library (without an extension).
makeInfo.library(n).Location	A character array that specifies the folder in which the library is located when precompiled. See the description of makeInfo.precompile in the preceding table for more information. A target can use the TargetPreCompLibLocation parameter to override this value. See “Specify the Location of Precompiled Libraries” for details.
makeInfo.library(n).Modules	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,'somdir2','lib');
   makeInfo.library(1).Modules  = {'srcfile1' 'srcfile2' 'srcfile3' };
```

Note: If a path that you specify in the rtwmakecfg.m API contains spaces, the code generator does not automatically convert the path to its non-space equivalent. If the build environments you intend to support do not support spaces in paths, refer to “Enable Build When Path Names Contain Spaces”.

Modify the Template Makefile

To expand the information generated by an `rtwmakecfg` function, you can modify the following sections of your target's TMF:

- `Include Path`
- `C Flags` and/or `Additional Libraries`
- `Rules`

The TMF code examples below may not apply to your make utility. For additional examples, see the GRT or ERT TMFs located in *`matlabroot`*`/rtw/c/grt/*.tmf` or *`matlabroot`*`/rtw/c/ert/*.tmf`.

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, as shown below.

```
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 on how to use configuration parameters to control library names and location during the build process, see “Control Library Location and Naming During Build” on page 24-7.

Add Rules to the Makefile

The following 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<|
```

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 Simulink Coder code generator 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 following procedure explains these steps in more detail.

1 Set the library file suffix, including the file type extension, based on the platform in use.

   Consider checking for the type of platform in use and then using the `TargetLibSuffix` parameter to set the library suffix accordingly. For example, you might set the suffix to `.a` for a UNIX platform and `_vc.lib` otherwise.

```
  if isunix
    suffix = '.a';
  else
    suffix = '_vc.lib';
  end

set_param(my_model,'TargetLibSuffix', suffix);
```

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 24-9.
   - Set the `makeInfo.precompile` field in an `rtwmakecfg.m` function file.

   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 code generator automatically 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 you can define in the structure. These fields are optional, except `rtwmakecfgDirs`.

Field	Description
`rtwmakecfgDirs`	A cell array of strings that name the folders containing `rtwmakecfg` files for libraries to be precompiled. The function uses the `Name` and `Location` elements of `makeInfo.library`, as returned by `rtwmakecfg`, to specify the name and location of the precompiled libraries. If you set the `TargetPreCompLibLocation` parameter to specify the library folder, that setting overrides the `makeInfo.library.Location` setting. **Note:** The specified model must contain blocks that use precompiled libraries specified by the `rtwmakecfg` files because the TMF-to-makefile conversion generates the library rules only if the libraries are used.
`libSuffix`	A string that specifies the suffix, including the file type extension, to be appended to the name of each library (for example, `.a` or `_vc.lib`). The string must include a period (.). You must set the suffix with either this field or the `TargetLibSuffix` parameter. If you specify a suffix with both mechanisms, the `TargetLibSuffix` setting overrides the setting of this field.
`intOnlyBuild`	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.
`makeOpts`	A string that specifies an option to be included in the `rtwMake` command line.
`addLibs`	A cell array of structures that specify libraries to be built that are not specified by an `rtwmakecfg` function. Each structure must be defined with two fields that are character arrays: • `libName` — the name of the library without a suffix • `libLoc` — the location for the precompiled library The TMF can specify other libraries and how those libraries are to be built. Use this field if you need to precompile those libraries.

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);
```

Program Building, Interaction, and Debugging

- “Compiler or IDE Selection and Configuration” on page 17-2
- “Program Builds” on page 17-11
- “Build and Run a Program” on page 17-44
- “Profile Code Performance” on page 17-46
- “Data Exchange” on page 17-50

Compiler or IDE Selection and Configuration

In this section...

"Compilers and the Build Process" on page 17-2
"Language Standards Compliance" on page 17-3
"Language Considerations" on page 17-3
"C++ Language Limitations" on page 17-4
"International Character Support" on page 17-5
"Choose and Configure Compiler on Microsoft Windows" on page 17-7
"Choose and Configure Compiler on UNIX" on page 17-7
"Include S-Function Source Code" on page 17-8
"Troubleshoot Compiler Configurations" on page 17-8

Compilers and the Build Process

The Simulink Coder 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`

Most Simulink Coder targets create an executable that runs on your workstation. When creating the executable, the Simulink Coder 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 Targets Available from the System Target File Browser.

For both Simulink Coder generated files and user-supplied files, the file extension, `.c` or `.cpp`, determines whether a C or a C++ compiler will be used in the Simulink Coder build process. If the file extension is `.c`, a C compiler will be used to compile the file,

and the symbols will use the C linkage convention. If the file extension is `.cpp`, a C++ compiler will be used to compile the file, and the symbols by default will use the C++ linkage specification.

Language Standards Compliance

The Simulink Coder software generates 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 generated by the Simulink Coder software from the following sources is ANSI C/C++ compliant:

- Simulink built-in block algorithmic code
- Simulink Coder and Embedded Coder 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 Simulink Coder software 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 Simulink Coder software, and can be a source for compliance problems, such as code that uses C99 features not supported in the ANSI C, C89/C90 subset.

Language Considerations

Simulink Coder supports C and C++ code generation. Consider the following as you choose a language for your generated code:

- Whether you need to configure Simulink Coder to use a specific compiler. This is required to generate C++ code on Windows. See “Compiler or IDE Selection and Configuration” on page 17-2.
- The language configuration setting for the model. See “Change Programming Language” on page 10-46.
- Whether you need to integrate legacy or custom code with generated code. For a summary of integration options, see “Integration Options”.
- Whether you need to integrate C and C++ code. If so, see “Integration Options”.

  Note: You can mix C and C++ code when integrating Simulink Coder 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 17-4.

For an example, enter sfcndemo_cppcount in the MATLAB Command Window. For a Stateflow example, enter sf_cpp.

C++ Target Language Considerations

To use the C++ target language support, you might need to configure the Simulink Coder software 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 Simulink Coder software to use a C++ compiler before you specify C++ for code generation, the software produces an error message.

C++ Language Limitations

- Simulink Coder does not support C++ code generation for the following:
  SimDriveline
  SimMechanics
  SimPowerSystems
  Embedded Targets in Embedded Coder
  Simulink Real-Time
- The following Embedded Coder dialog box fields currently do not accept the `.cpp` extension. However, a `.cpp` file will be generated 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 an **mpt data object** in Model Explorer

These restrictions on specifying `.cpp` will be removed in a future release.

International Character Support

The code generator does not include non-US-ASCII characters in compilable portions of generated source code. However, Simulink, Stateflow, Simulink Coder, and Embedded Coder support non-US-ASCII characters in certain ways. When the code generator encounters non-US-ASCII characters, the characters become comments in the generated code or do not propagate into the generated source files.

You can include non-US-ASCII characters in these elements:

Source	Simulink Coder	Embedded Coder
Block names[1, 2, 3]	X	X
User comments in Stateflow diagrams	X	X
Custom TLC files (`.tlc`)[4]	X	X
Block descriptions that you enter from the Simulink Block Parameter dialog box		X
Comments in code generation template files (`.cgt`) [5]		X
Stateflow object descriptions		X
Simulink and mpt data object descriptions		X

[1] You cannot include non-US-ASCII characters in block names for nonvirtual subsystems configured to use the subsystem name as the function or file name.

[2] If the code generator uses a block name as an identifier in generated code, the identifier includes only US-ASCII characters.

[3] If you want the code generator to produce code that includes support for data logging during execution, use the ASCII-7 character set when naming blocks. Otherwise, the block names in block paths included in the log are not readable and do not compile.

Source	Simulink Coder	Embedded Coder
[4] You cannot use non-US-ASCII characters in TLC variable or function names.		
[5] TLC comments in the code generation templates files can include non-US-ASCII characters. However, the code generator does not propagate the comments to the generated code.		

You can configure the code generator to include the text associated with these elements in generated code comments by using parameters on the **Code Generation** > **Comments** pane of the Configuration Parameters dialog box. For example, to configure the code generator to include block names in block code comments, select **Simulink block / Stateflow object comments**. For more information, see "Configure Code Comments".

Use the Simulink Model Advisor check "Check model for foreign characters" to check a model for characters that the code generator cannot represent in the current encoding.

Character Set Limitation

You can encounter problems with models containing characters of a specific character set, such as Shift JIS, on a host system for which that character set is not configured as the default.

When models containing characters of a given character set are used on a host system that is not configured with that character set as the default, Simulink can incorrectly interpret characters during model loading and saving. This can lead to corrupted characters being displayed in the model and possibly the model failing to load. It can also lead to corrupted characters in the model file if you save it.

This limitation does not exist when the characters used in the model are in the default character set for the host system. For example, you can use Shift JIS characters without issues if the host system is configured to use Japanese Windows.

Additionally, during code generation, the Target Language Compiler can have similar problems reading characters from either the `*model*.rtw` or user written `.tlc` files. This can result in corrupt characters in generated source file comments or a Target Language Compiler error.

For an example of international character set support for code generation, run the example model `rtwdemo_international`. This example model is set up to work around the character limitations described above. If you run this example from a non-Japanese

MATLAB host machine, you must set up an international character set for Simulink. For example, type

```
bdclose all; set_param(0, 'CharacterEncoding', 'Shift_JIS')
rtwdemo_international
```

Other uses of non-US-ASCII characters in models or in files used during the build process are not supported; you should not depend on any incidental functionality that may exist.

For additional information, see the `slCharacterEncoding` function.

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 Simulink Coder build process. Alternatively, you can use the `lcc` C compiler that is shipped with the MATLAB product.

The compiler that the Simulink Coder code generator requires is determined primarily by the **System target file** parameter and the **Build process** parameters located on the **Code Generation** pane of the Configuration Parameters dialog box:

- If you select a toolchain-based system target file such as `grt.tlc` (Generic Real-Time Target) or `ert.tlc` (Embedded Coder), the **Build process** subpane displays toolchain parameters for configuring the build process. Use the **Toolchain** parameter to select and validate a compiler and associated tools for your model build.
- If you select a template makefile (TMF) based system target file, such as a desktop or embedded target, `autosar.tlc`, `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 Targets” 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 Simulink Coder 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) or `ert.tlc` (Embedded Coder), the **Build process** subpane displays toolchain parameters for configuring the build process. You can use the toolchain parameters 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, `autosar.tlc`, 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 Targets" for valid UNIX TMF names.) You can display the UNIX compiler with the following command:

  ```
  mex -setup
  ```

Include S-Function Source Code

When the Simulink Coder 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 *`sfncname`*`.h` in the same folder that the S-function MEX-file is in. This folder must be on the MATLAB path.

Similarly, the Simulink Coder code generator adds a rule for the folder when it finds a file *`sfncname`*`.c` (or `.cpp`) in the same folder as the S-function MEX-file is in.

Troubleshoot Compiler Configurations

- "Compiler Version Mismatch Errors" on page 17-9
- "Generated Executable Image Produces Incorrect Results" on page 17-9
- "Compile-Time Errors" on page 17-10

Compiler Version Mismatch Errors

Explanation

You received a version mismatch error when you compiled code generated by the Simulink Coder software.

User Action

1. Check the list of currently supported and compatible compilers available at http://www.mathworks.com/support/compilers/current_release/.
2. If necessary, upgrade or change your compiler. For more information, see "Choose and Configure Compiler on Microsoft Windows" on page 17-7 or "Choose and Configure Compiler on UNIX" on page 17-7.
3. Rebuild the model.

Generated Executable Image Produces Incorrect Results

Explanation

You applied compiler optimizations when you used Simulink Coder to generate an executable image. However, the optimizations caused the executable image to produce incorrect results, even though expected code was generated.

User Action

Do one of the following:

- Lower the compiler optimization level.
  1. Select `Custom` for the Model Configuration parameter **Code Generation** > **Compiler optimization level**. The **Custom compiler optimization flags** field appears.
  2. Specify a lower optimization level in the **Custom compiler optimization flags** field.
  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” and your compiler documentation.

Compile-Time Errors

Explanations

- You received a compiler configuration error.
- Environment variables for your make utility, compiler, or linker are set up incorrectly. For example, installation of Cygwin tools on a Windows platform might affect environment variables used by other compilers.
- Custom code specified as an S-function block or in the **Code Generation** > **Custom Code** pane of the Configuration Parameters dialog includes errors. For example, the code might refer to a header file that the compiler cannot find.
- The model includes a block, such as a device driver block, that is not intended for use with the currently selected target.

User Actions

- Make sure that MATLAB supports the compiler and version that you want to use. For a list of currently supported and compatible compilers, see http://www.mathworks.com/support/compilers/current_release/. If necessary, upgrade or change your compiler (see “Choose and Configure Compiler on Microsoft Windows” on page 17-7 or “Choose and Configure Compiler on UNIX” on page 17-7).
- Review the environment variable settings for your system by using the `set` command on a Windows platform or `setenv` on a UNIX platform. Make sure the settings match what is required for the tools you are using.
- Remove the custom code from the model, to help isolate the source of the problem, debug, and rebuild.
- Remove the target-specific block or configure the model for use with the another target.

Program Builds

In this section...

"Configure the Build Process" on page 17-11
"Initiate the Build Process" on page 17-18
"Build a Generic Real-Time Program" on page 17-19
"Rebuild a Model" on page 17-28
"Control Regeneration of Top Model Code" on page 17-29
"Reduce Build Time for Referenced Models" on page 17-31
"Relocate Code to Another Development Environment" on page 17-35
"How Executable Programs Are Built From Models" on page 17-40

Configure the Build Process

- "Choose a Build Process" on page 17-11
- "Toolchain Approach" on page 17-12
- "Upgrade Model to Use Toolchain Approach" on page 17-13
- "Template Makefile Approach" on page 17-16
- "Specify TLC Options" on page 17-18

Choose a Build Process

Simulink Coder software provides two separate build processes for building code generated from Simulink models:

- *Toolchain approach* — A newer build process that generates optimized makefiles and supports custom toolchains
- *Template makefile approach* — An older build process that uses template makefiles

The **System target file** parameter, located on the **Code Generation** pane of the Configuration Parameters dialog box, determines which build process is used for a model. When the **System target file** is set to:

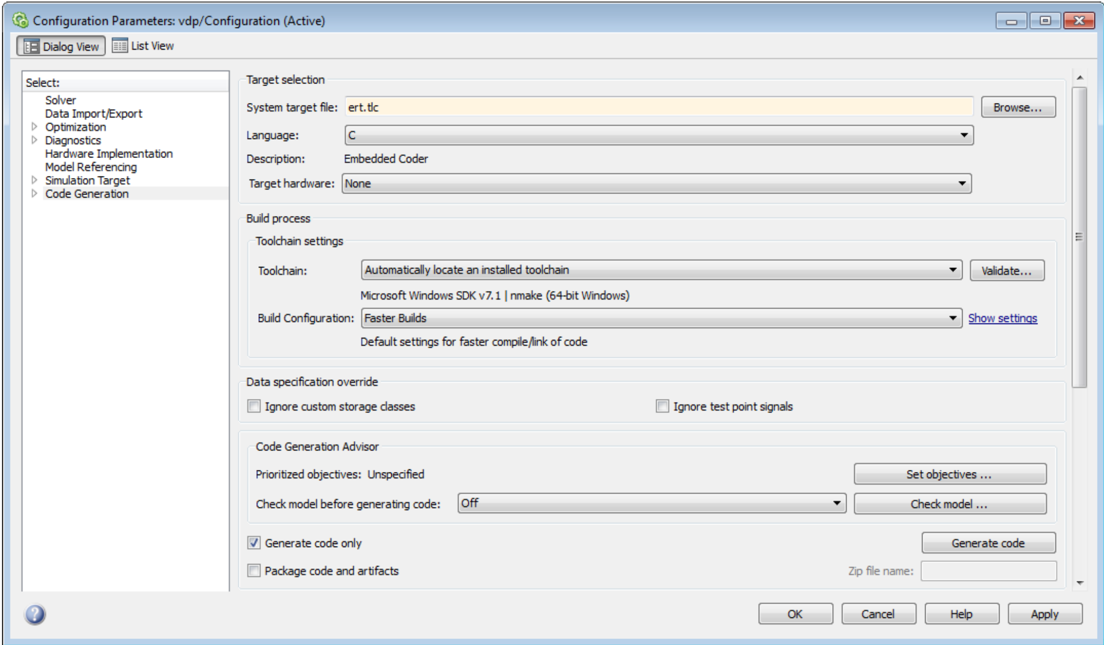
- `ert.tlc`, `ert_shrlib.tlc`, or `grt.tlc`, the build process uses the *toolchain approach*. For more information, see "Toolchain Approach" on page 17-12.

- Other `*.tlc` files, the build process uses the *template makefile approach*. For more information, see “Template Makefile Approach” on page 17-16.

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)



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 **Validate** button to check that the toolchain is present, and that Simulink Coder software 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, use the **Browse** button 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`

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.

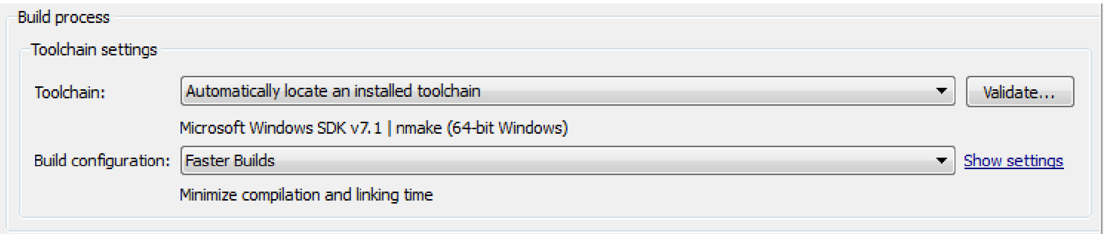
We recommend upgrading your models, but doing so is not required. You can continue generating code from a model that has not been upgraded.

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.

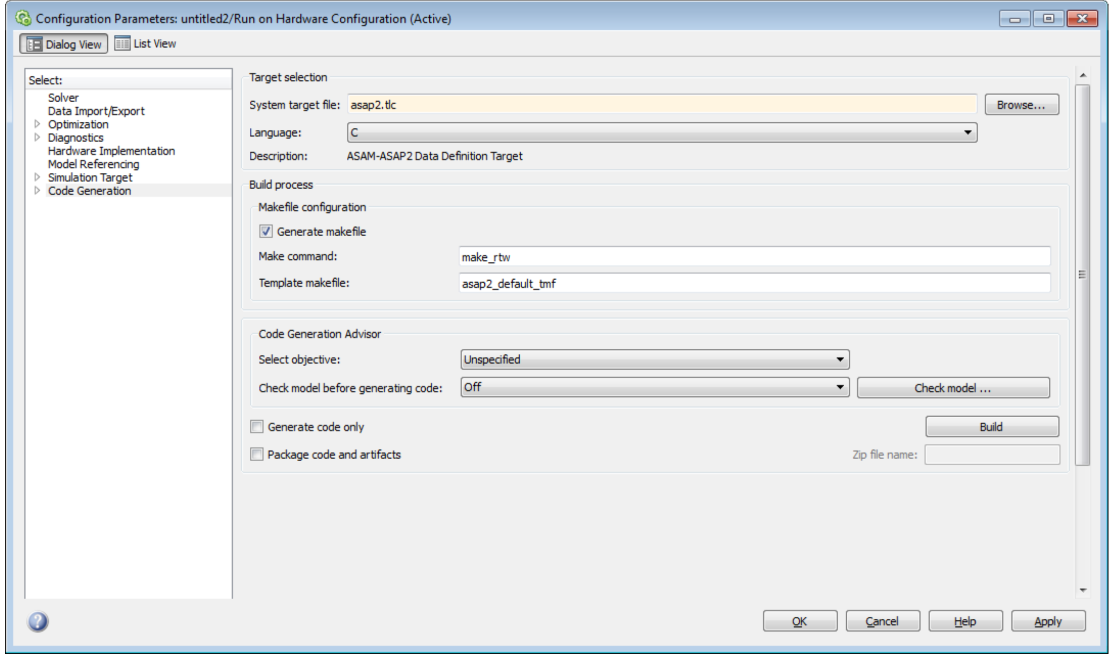
Note: After registering the custom toolchain, update **Toolchain** to use the custom toolchain.

---

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 Simulink Coder build process is to generate a makefile for a model. By default, the Simulink Coder build process generates

a makefile. You can suppress the 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 option,

- 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 24-14.

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 10-35 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 "Code Generation Pane: General", 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 10-35 gives a detailed description of the logic by which the Simulink Coder 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 Simulink Coder 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”.

Initiate the Build Process

You can initiate code generation and the build process by using the following options:

- Clear the **Generate code only** option on the **Code Generation** pane of the Configuration Parameters dialog box and click **Build**.
- 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.

Build a Generic Real-Time Program

- "Building a Program" on page 17-19
- "Working Folder" on page 17-19
- "Build and Code Generation Folders" on page 17-20
- "Set Simulation Parameters" on page 17-20
- "Select a Target Configuration" on page 17-21
- "Set Code Generation Parameters" on page 17-22
- "Build and Run a Program" on page 17-26
- "Contents of the Build Folder" on page 17-27

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 and Code Generation Folders

During code generation, the Simulink Coder software 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 working folder to organize code for the referenced models. Code generation folders exist alongside of Simulink Coder build folders, and are named `slprj`. "Generate Code for Referenced Models" on page 4-4 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 Simulink Coder software can generate code for models using variable-step solvers for rapid simulation (`rsim`) and S-function targets only.

To set simulation parameters using the Configuration Parameters dialog box:

1 Open the `myAircraftExample` model if it is not already open.

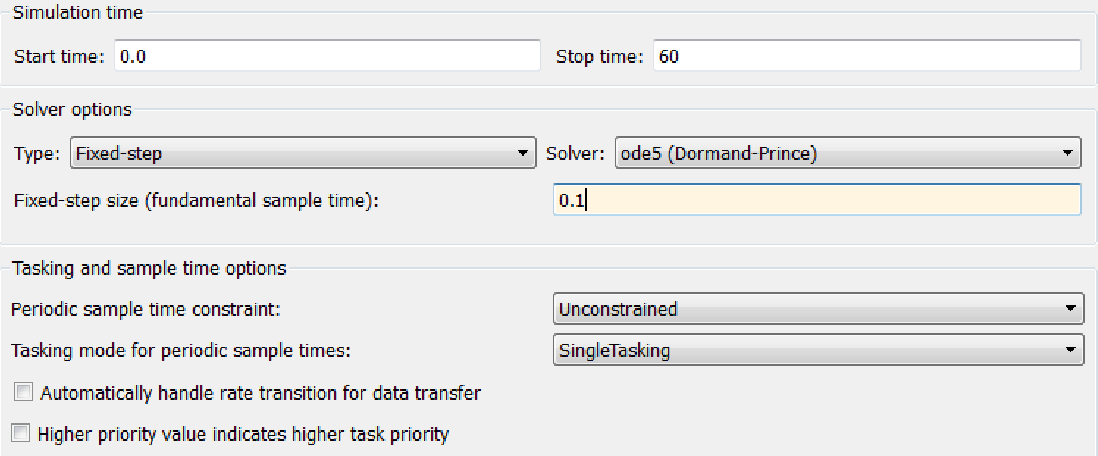
2 From the model window, open the Configuration Parameters dialog box by selecting **Simulation** > **Model Configuration Parameters**.

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`
- **Tasking mode for periodic sample times**: `SingleTasking`

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

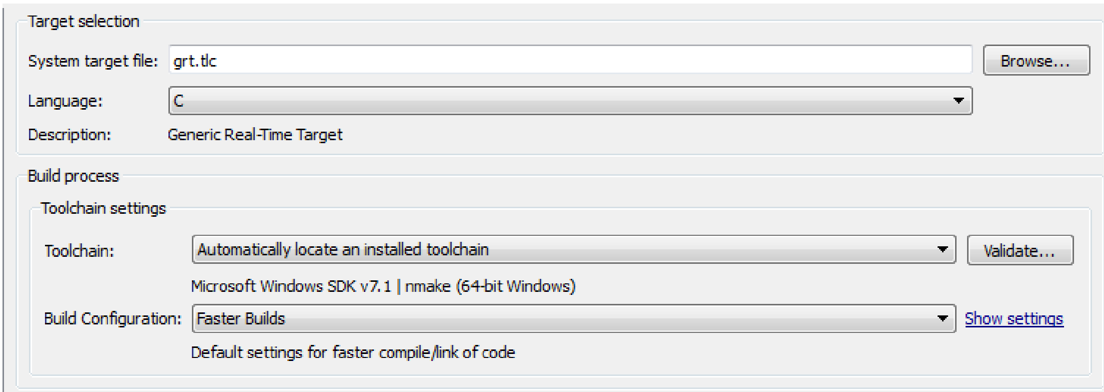
To specify a target configuration for your model, you choose a system target file, a toolchain or template makefile, and a `make` command.

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 section do not require you to make changes. They are included to help you familiarize yourself with the Simulink Coder 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.

☑ Create code generation report ☑ Open report automatically

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.

Auto-generated identifier naming rules

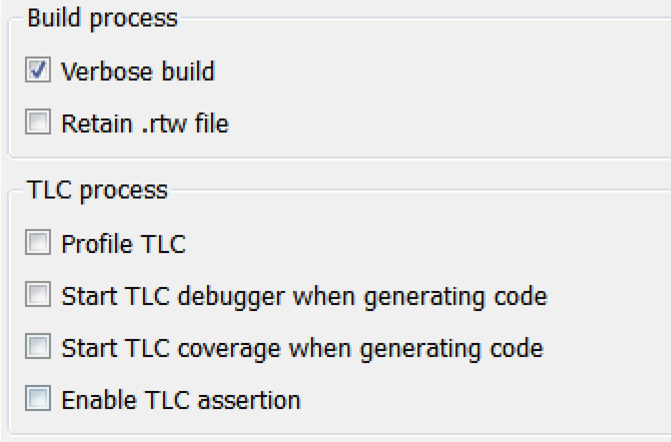
Maximum identifier length: 31

Reserved names

☐ Use the same reserved names as Simulation Target

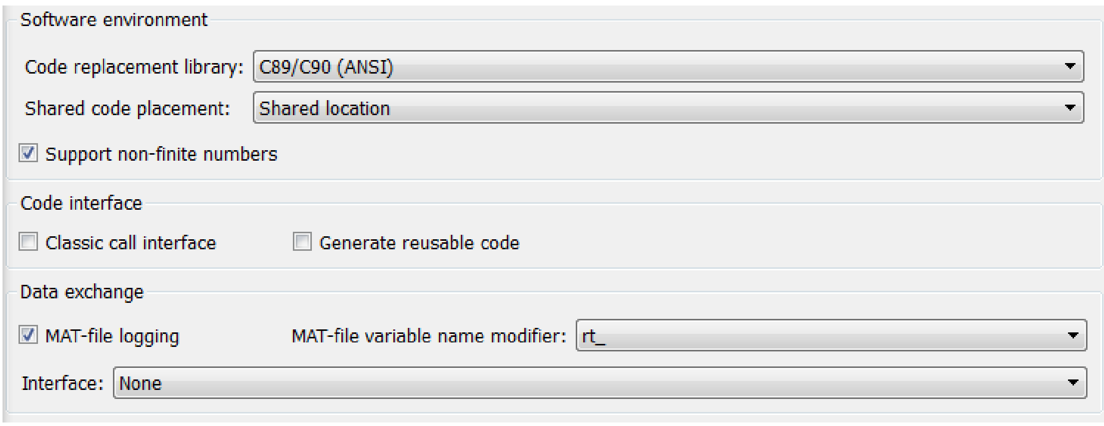
Reserved names:

5 Select the **Code Generation** > **Debug** pane. The options displayed here control build verbosity and debugging support, and are common to all target configurations. 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 working folder.
- If it is not already cleared, clear the **Classic call interface** option.



7 Click **Apply** and save the model.

Build and Run a Program

The Simulink Coder 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 working folder now contains an executable, `myAircraftExample.exe` (Microsoft Windows platforms) or `myAircraftExample` (UNIX platforms). In addition, the Simulink Coder build process has created a code generation folder, `slprj`, and a build folder, `myAircraftExample_grt_rtw`, in your working 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.slx          slprj
   ..                         myAircraftExample.slx.autosave
   myAircraftExample.exe      myAircraftExample_grt_rtw
   ```

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
`myAircraftExample.c`	Standalone C code that implements the model
`myAircraftExample.h`	An include header file containing definitions of parameters and state variables
`myAircraftExample_private.h`	Header file containing common include definitions
`myAircraftExample_types.h`	Forward declarations of data types used in the code
`rtmodel.h`	Master header file for including generated code in the static main program (its name does not change, and it simply includes `myAircraftExample.h`)

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:

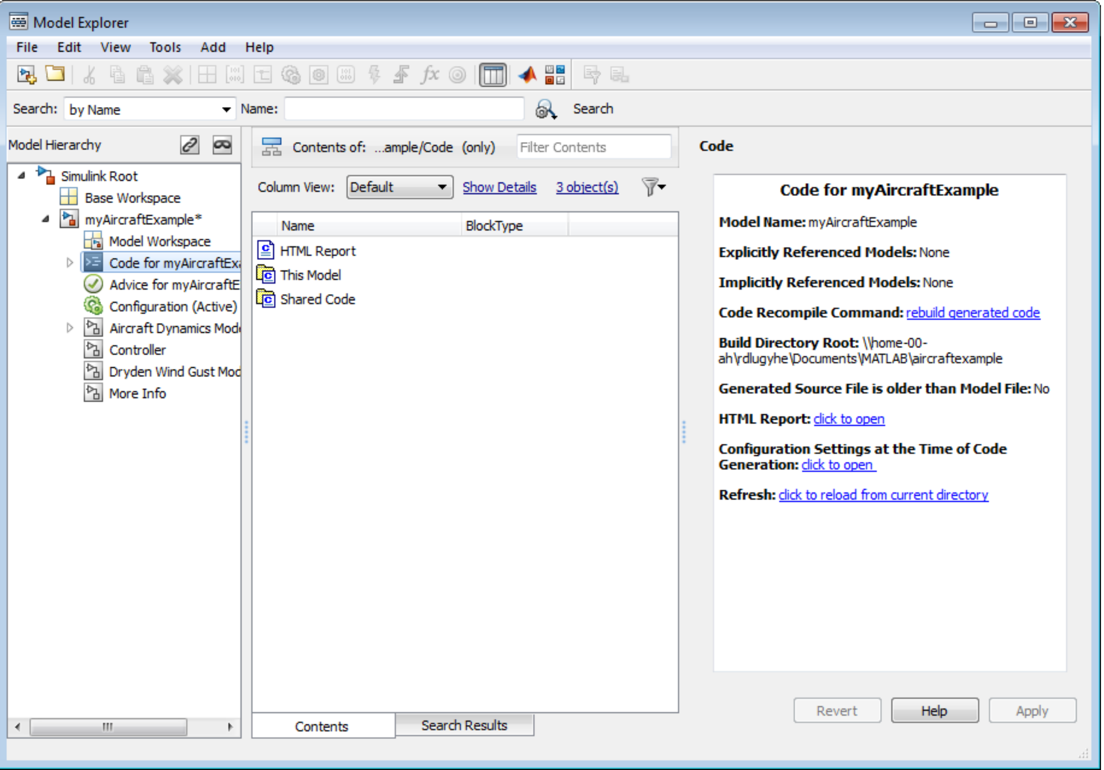
- myAircraftExample.mk — Makefile for building executable using the specified Toolchain.
- Object (.obj) files
- myAircraftExample.bat — Batch control file
- rtw_proj.tmw — Marker file
- buildInfo.mat — Build information for relocating generated code to another development environment
- defines.txt — 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”.

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**.



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 **Code Generation** > **Debug** pane 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 17-30.

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 24-25.

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 code generation folders, for example, `slprj` or the generated model code folder.

Reduce Build Time for Referenced Models

- "Parallel Building For Large Model Reference Hierarchies" on page 17-31
- "Parallel Building Configuration Requirements" on page 17-32
- "Build Models In a Parallel Computing Environment" on page 17-32
- "Locate Parallel Build Logs" on page 17-34

Parallel Building For Large Model Reference Hierarchies

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™ (MDCS) software, code generation and compilation for each referenced model can be distributed 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, 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 17-32.

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 17-32.

For information on how to configure a custom embedded target to support parallel builds, see “Support Model Referencing” on page 26-78.

Note: In an MDCS parallel computing configuration, parallel building is designed to work interactively with the MATLAB Distributed Computing Server 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:

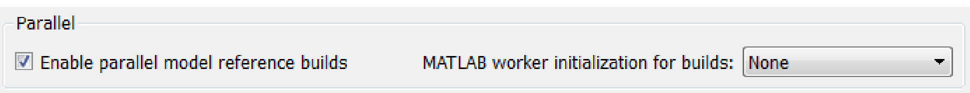
- For local pools, the host machine must have enough RAM to support the number of local workers (MATLAB sessions) that you plan to use. For example, using `parpool(4)` to create a parallel pool with four workers results in five MATLAB sessions on your machine, each using approximately 120 MB of memory at startup.
- Remote MDCS workers participating in a parallel build 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 In 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).

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 **Optimization** pane 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 17-34.

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_model001'
### 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`.

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_mdl_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_mdl_ref/bot_model001/slprj/grt/bot_model001`, *`build_folder`*`/par_mdl_ref/bot_model001/slprj/ert/bot_model001`, or *`build_folder`*`/par_mdl_ref/bot_model001/slprj/sim/bot_model001`.

Relocate Code to Another Development Environment

- "Code Relocation" on page 17-35
- "Package Code Using the Graphical User Interface" on page 17-36
- "Package Code Using the Command-Line Interface" on page 17-36
- "packNGo Function Limitations" on page 17-40

Code Relocation

If you need to relocate the static and generated code files for a model to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB and Simulink products, use the Simulink Coder pack-and-go utility. This 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:

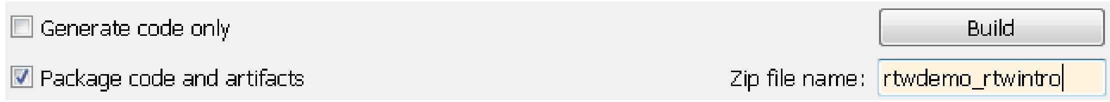
- Use the model option "**Package code and artifacts**" on the **Code Generation** pane of the Configuration Parameters dialog box. See "Package Code Using the Graphical User Interface" on page 17-36.

- 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 17-36. 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 `.` 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 Simulink Coder 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, as described in "Customize Post-Code-Generation Build Processing" on page 24-14.

  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	`packNGo(buildInfo, {'packType' 'hierarchical'});`
Rename the primary `zip` file	`packNGo(buildInfo, {'fileName' 'zippedsrcs'});`
Change the structure of the file packaging to hierarchical and rename the primary `zip` file	`packNGo(buildInfo, {'packType' 'hierarchical'... 'fileName' 'zippedsrcs'});`
Include header files found on the include path in the `zip` file	`packNGo(buildInfo, {'minimalHeaders' false});`
Generate warnings for parse errors and missing files	`packNGo(buildInfo, {'ignoreParseError' true...`

To...	Specify...
	'ignoreFileMissing' true});

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_rtwintro using the command-line interface:

1. Set your working folder to a writable folder.
2. Open the model rtwdemo_rtwintro and save a copy to your working folder.
3. Enter the following MATLAB command:

   ```
   set_param('rtwdemo_rtwintro', '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.

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.

How Executable Programs Are Built From Models

- "Build Process Steps" on page 17-40
- "Customized Makefile Generation" on page 17-41
- "Executable Program Generation" on page 17-41

Build Process Steps

The Simulink Coder software generates C code only or generates the C code and produces an executable image, depending on the level of processing you choose. By default, a **Build** button appears on the **Code Generation** pane of the Configuration Parameters dialog box. This button completes the entire build process and an executable image results. If you select the **Generate code only** check box to the left of the button, the button label changes to **Generate Code**.

When you click the **Build** or **Generate Code** button, the Simulink Coder software 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 11-63
2. "Code Generation" on page 11-63
3. "Customized Makefile Generation" on page 17-41
4. "Executable Program Generation" on page 17-41

For more information, see "Generate a Code Generation Report". You can also view an HTML report in Model Explorer.

Customized Makefile Generation

After generating the code, the Simulink Coder software generates 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.

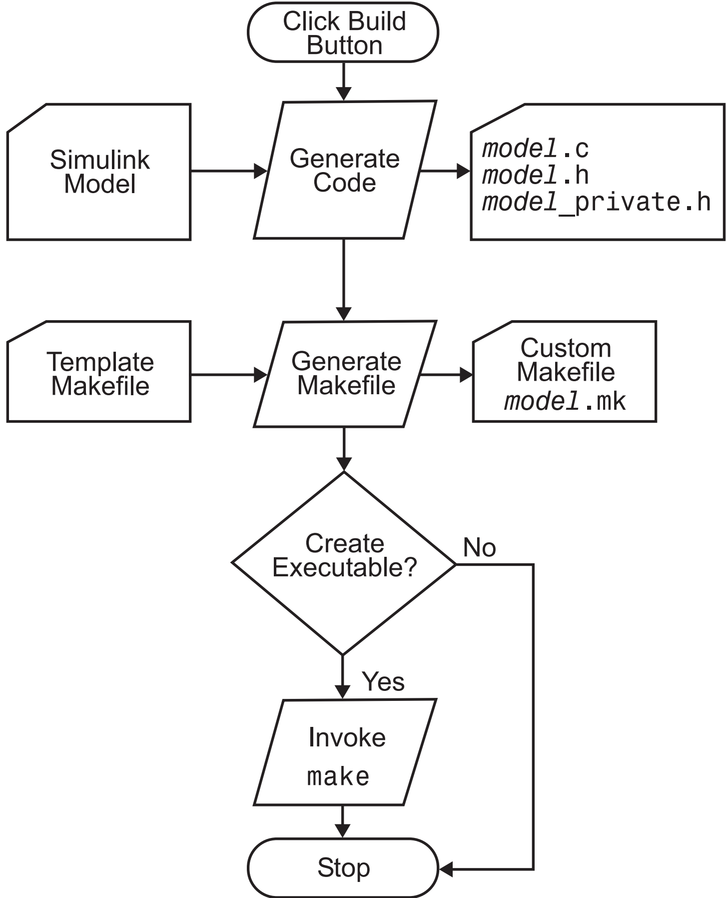
The Simulink Coder software creates `model.mk` from a system template file, `system.tmf` (where `system` stands for the selected target name). The system template makefile is designed for your target environment. You have the option of modifying the template makefile to specify compilers, compiler options, and additional information used during the creation of the executable.

The Simulink Coder software creates the `model.mk` file by copying the contents of `system.tmf` and expanding lexical tokens (symbolic names) that describe your model's configuration.

The Simulink Coder software provides many system template makefiles, configured for specific target environments and development systems. "Select a Target" in the Simulink Coder documentation lists the template makefiles that are bundled with the Simulink Coder software. To see an example template makefile, navigate to `matlabroot/rtw/c/grt`, and open with an editor the file `grt_msvc.tmf`. You can fully customize your build process by modifying an existing template makefile or providing your own template makefile.

Executable Program Generation

The following figure shows how the Simulink Coder software controls automatic program building.

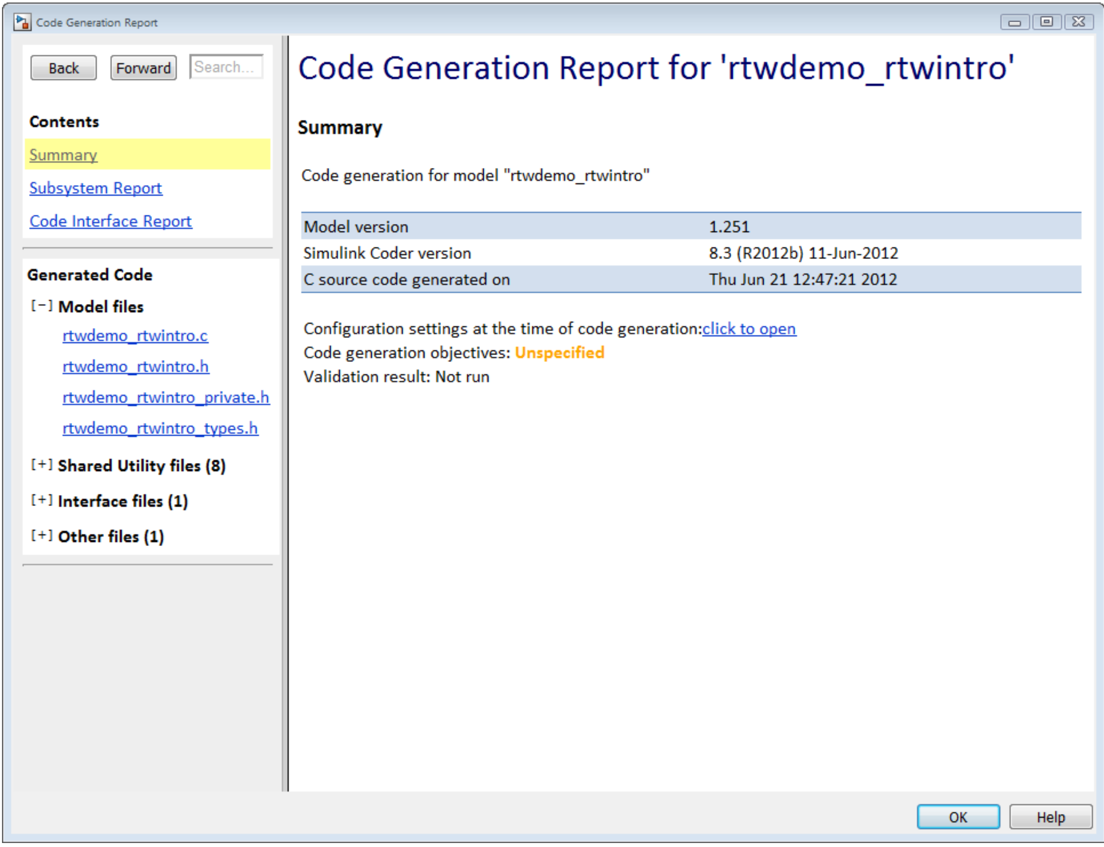


During the final stage of processing, the Simulink Coder build process invokes the generated makefile, `model.mk`, which in turn compiles and links the generated code. On PC platforms, a batch file is created to invoke the generated makefile. The batch file sets up the environment for invoking the `make` 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.



Build and Run a Program

The Simulink Coder 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`
     - **Tasking mode for periodic sample times**: `SingleTasking`
   - Select the **Code Generation** > **Report** pane, and select the **Create code generation report** parameter. This causes the Simulink Coder 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 your working folder.
     - If it is not already cleared, clear the **Classic call interface** 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 slexAircraftExample.exe
### Successful  completion of build procedure for model: myAircraftExample
### Creating HTML report file myAircraftExample_codegen_rpt.html
```

The working folder now contains an executable, `myAircraftExample.exe` (Microsoft Windows platforms) or `myAircraftExample` (UNIX platforms). In addition, the Simulink Coder build process has created a code generation folder, `slprj`, and a build folder, `myAircraftExample_grt_rtw`, in your working folder.

Note: The Simulink Coder 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 `!slexAircraftExample`. The ! character passes the command that follows it to the operating system, which runs the standalone `slexAircraftExample` 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
```

Profile Code Performance

In this section...

"About Profiling Code Performance" on page 17-46
"How to Profile Code Performance" on page 17-46
"Run Profiling Hooks for Generated Code" on page 17-49
"Profiling Limitation" on page 17-49

About Profiling Code Performance

By profiling the performance of generated code, you can help verify that the code meets performance requirements. 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

Function	Input Arguments	Output Type	Description
			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.
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.

Function	Input Arguments	Output Type	Description
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 17-49. For details on programming a .tlc file and defining TLC configuration variables, see “Target Language Compiler”.

Run Profiling Hooks for Generated Code

The example Profiling Hooks for Generated Code shows how to use special hooks provided by Simulink Coder 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 17-46. 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`).

Data Exchange

In this section...

"Host/Target Communication" on page 17-50
"Logging" on page 17-99
"Parameter Tuning" on page 17-110
"Data Interchange Using the C API" on page 17-125
"ASAP2 Data Measurement and Calibration" on page 17-158
"Direct Memory Access to Generated Code" on page 17-171

Host/Target Communication

- "About Host/Target Communication" on page 17-50
- "Set Up an External Mode Communication Channel" on page 17-51
- "Configure and Use External Mode" on page 17-62
- "External Mode Compatible Blocks and Subsystems" on page 17-78
- "External Mode Communication" on page 17-80
- "Choose Communication Protocol for Client and Server" on page 17-83
- "Use External Mode Programmatically" on page 17-92
- "Generate External Mode and C API Data Interfaces" on page 17-96
- "External Mode Limitations" on page 17-97

About Host/Target Communication

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 External mode, you can:

- 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 works by establishing 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 Wind River Systems Tornado target supports TCP/IP communication only.
- The Simulink Real-Time product uses a customized transport layer.
- The Real-Time Windows Target product uses shared memory communication.
- Target hardware platforms supported by Simulink use serial or TCP/IP communication.

Set Up an External Mode Communication Channel

- “External Mode Communication Channel Setup” on page 17-52
- “Set Up the Model” on page 17-52
- “Build the Target Executable” on page 17-54
- “Run the External Mode Target Program” on page 17-57
- “Tune Parameters” on page 17-61

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:

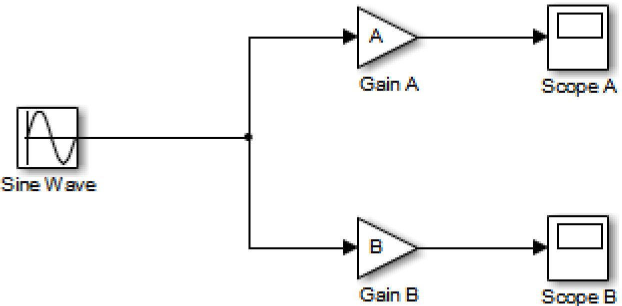
1. From the MATLAB command line, type:

   ```
   mkdir ext_mode_example
   ```

2. Make `ext_mode_example` your working folder:

   ```
   cd ext_mode_example
   ```

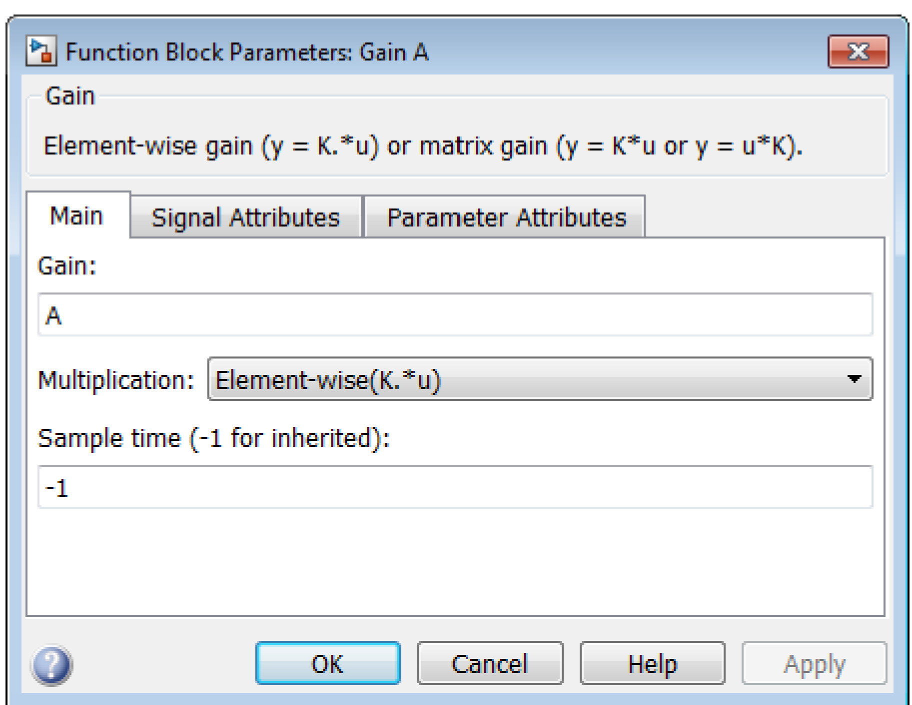
3. Create a model in Simulink with a Sine Wave block for the input signal, two Gain blocks in parallel, and two Scope blocks. Be sure to label the Gain and Scope blocks as shown below.



4 Define and assign two MATLAB workspace variables, A and B, as follows:

```
A = 2;
B = 3;
```

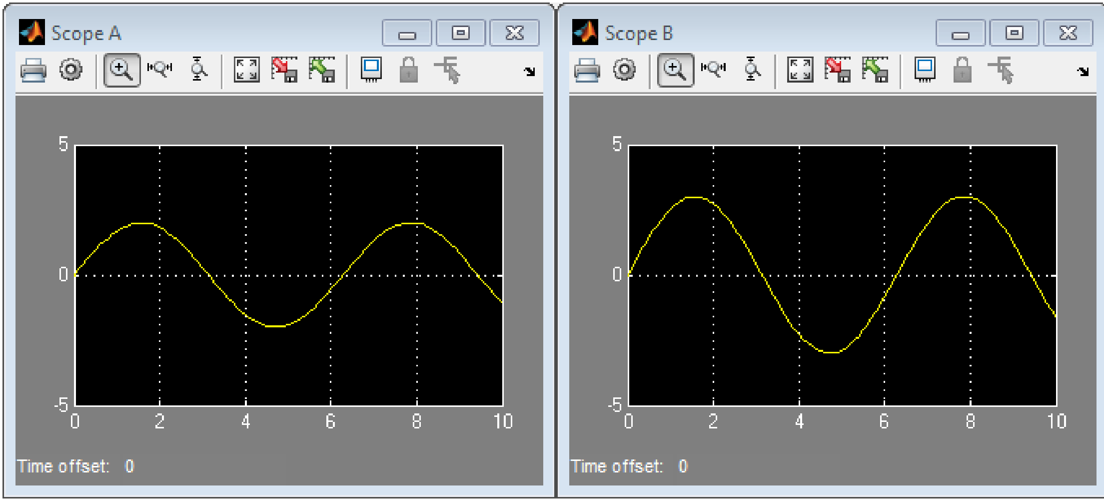
5 Open Gain block A and set its **Gain** parameter to the variable A.



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 17-61.

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. 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.)

Click **Apply**.

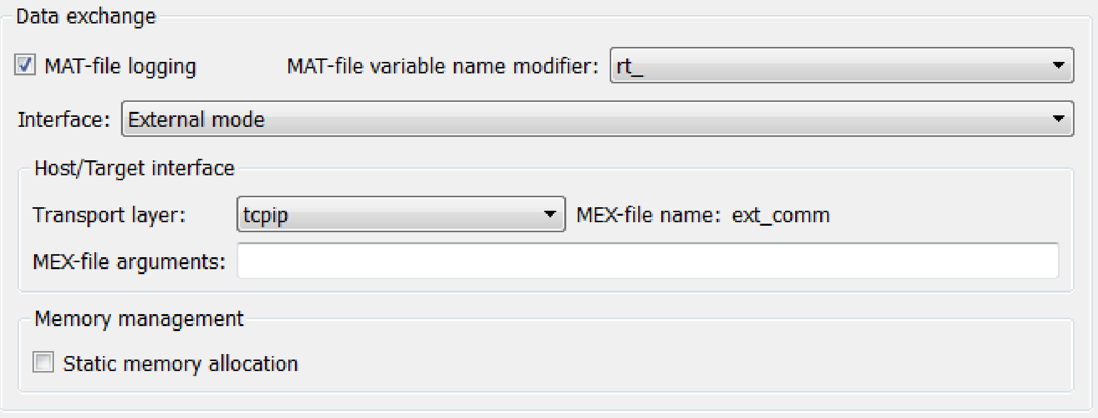
4. Select the **Data Import/Export** pane, and clear the **Time** and **Output** check boxes. In this example, data is not logged to the workspace or to a MAT-file. Click **Apply**.

5. Select the **Optimization** > **Signals and Parameters** pane. Make sure that the **Inline parameters** option is *not* selected. 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** section, from the **Interface** drop-down list, select `External mode`. This selection enables generation of External mode support code and reveals two more sections of controls: **Host/Target interface** and **Memory management**.

8. In the **Host/Target interface** section, make sure that the default value `tcpip` is selected for the **Transport layer** parameter. The **Data exchange** section now appears as shown below.



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”.

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” and “MEX-File Optional Arguments for Serial Transport” in the Simulink Coder documentation.

This example uses the default arguments. Leave the **MEX-file arguments** field blank.

9. Click **Apply** to save the **Interface** settings.
10. Save the model.
11. Select the **Code Generation** pane. Make sure that **Generate code only** is cleared, and then click **Build** to generate code and create the target program. The content of subsequent messages depends on your compiler and operating system. The final message is:

```
### Successful completion of build procedure for model: ex_extModeExample
```

Run the External Mode Target Program

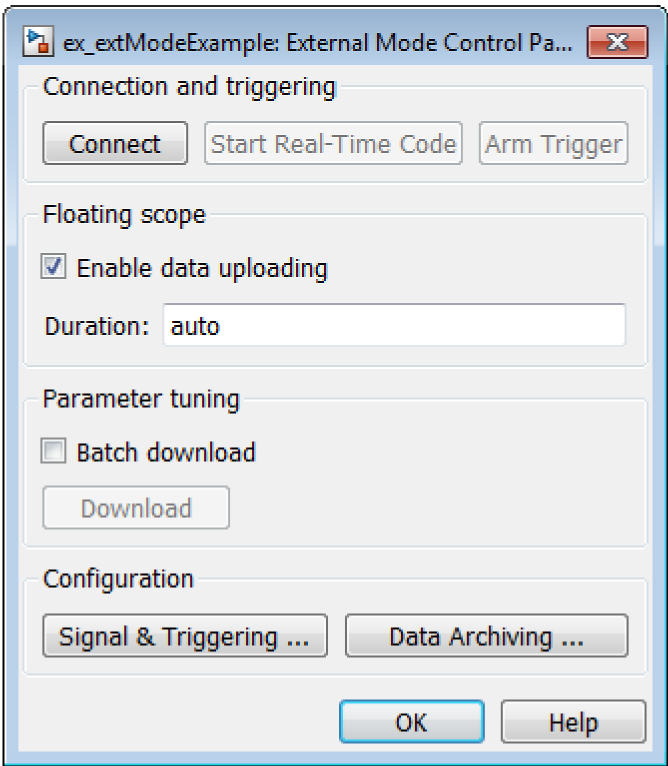
You now run the `ex_extModeExample` target executable and use Simulink as an interactive front end to the running target program. The executable file is in your working folder. Run the target program and establish communication between Simulink and the target.

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. Also, while the target program runs, you can use the dialog box to reconfigure signals.

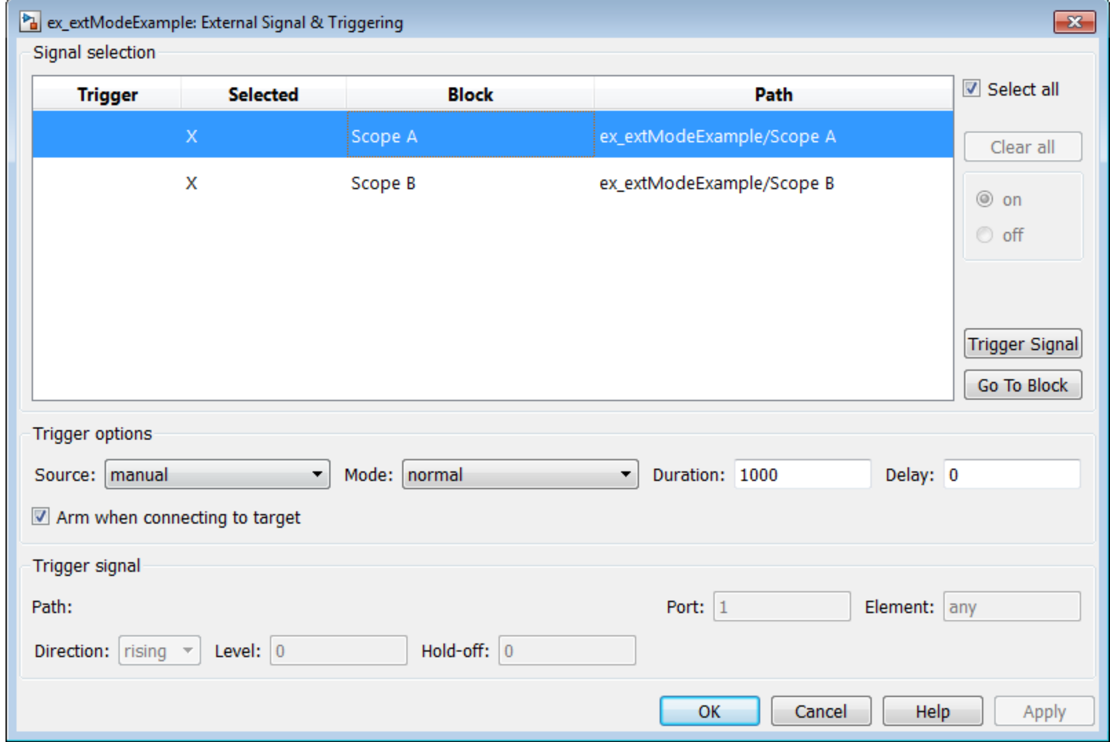
In this example, you observe and use the default settings of the External Signal & Triggering dialog box.

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 "Data Archiving" on page 17-75 for more information.

2. 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.



3 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 “External Signal Uploading and Triggering” on page 17-70.

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 Scope blocks A and B. 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 the you update the block diagram, the new values are downloaded to the target program.

To tune the variables `A` and `B`:

1. In the MATLAB Command Window, assign new values to both variables, for example:

   ```
   A = 0.5;
   B = 3.5;
   ```

2. Activate 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. You can also enter gain values directly into the Gain blocks. Open the Block Parameters dialog box for Gain block A or B in the model. Enter a new numerical value for the **Gain** parameter and click **Apply**. As soon as you click **Apply**, the new value is downloaded to the target program and the scope is changed because of the gain change.

Note: In the example model, you could change the frequency, amplitude, or phase of the sine wave signal by opening the Block Parameters dialog box for the Sine Wave block and entering a new numerical value. However, you cannot change the sample time of the Sine Wave block. 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.

4 To simultaneously disconnect host/target communication and end execution of the target program, from the **Simulation** menu, select **Stop**. Alternatively, you can go to the External Mode Control Panel and click **Stop Real-Time Code**.

Configure and Use External Mode

- "Configure External Mode Options for Code Generation" on page 17-62
- "Target Interfacing" on page 17-65
- "Control Host and Target Execution" on page 17-66
- "Control External Mode Operations" on page 17-67
- "Connecting, Starting, and Stopping" on page 17-68
- "Uploading Data to Floating Scopes" on page 17-69
- "Parameter Downloading" on page 17-69
- "External Signal Uploading and Triggering" on page 17-70
- "Data Archiving" on page 17-75

Configure External Mode Options for Code Generation

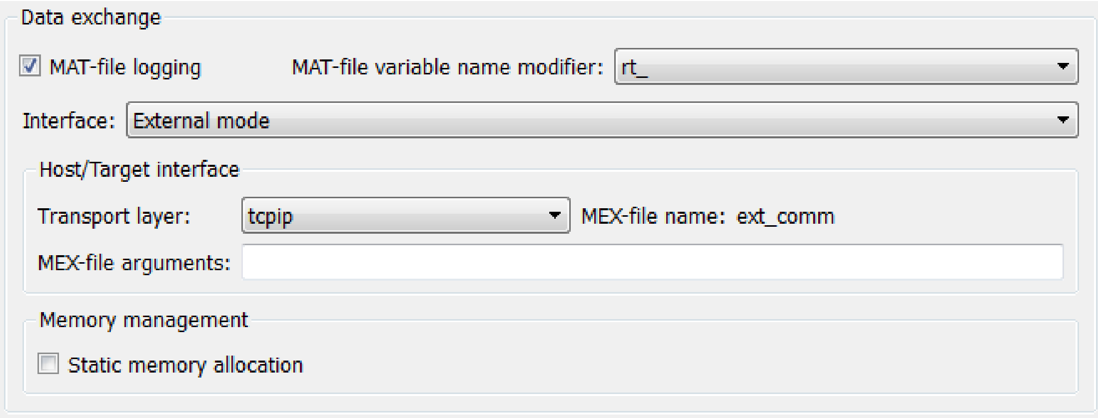
The list of targets and products that support External mode includes the ERT, GRT, RSim, and Tornado targets, the Real-Time Windows Target product, and Simulink target hardware platforms. Targets that support External mode provide a set of External mode options in the Configuration Parameters dialog 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:
  - In the **Code Generation** pane of the Configuration Parameters dialog box, click **Generate Code** or **Build**

- Press **Ctrl+B**
- At the MATLAB command line, enter `rtwbuild`

The following figure shows External mode parameters from the GRT target view of the **Code Generation** > **Interface** pane, **Data exchange** section.



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** section in the **Code Generation** > **Interface** pane includes the following External mode parameters:

- **Interface** menu: Selects a data exchange interface to include in the generated C code.

  When you select `External mode` from the Interface menu, 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 17-65 and "Choose Communication Protocol for Client and Server" on page 17-83.
- **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` from the **Interface** menu does not cause the Simulink model to operate in External mode (see "Control Host and Target Execution" on page 17-66). Its function is to instrument 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.

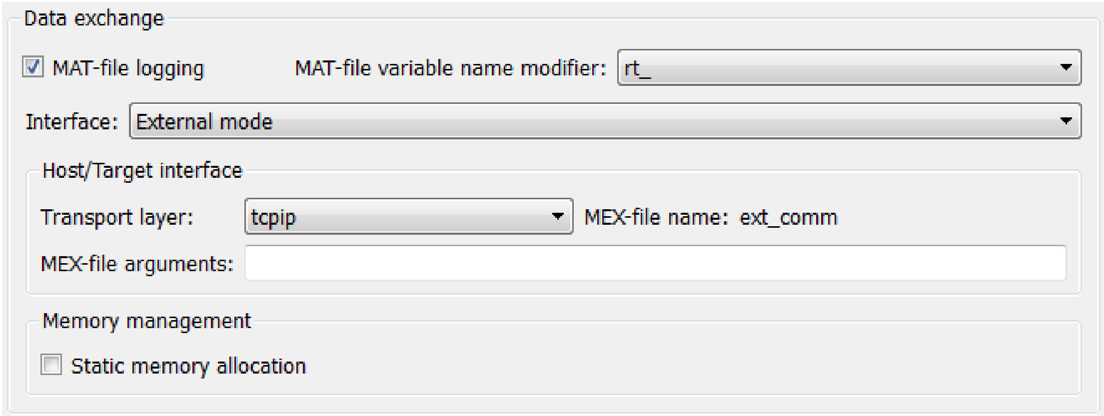
Note When you create an ERT target, External mode can generate pure integer code. Select pure integer code by clearing the **Support floating-point numbers** option on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box. Clearing this option makes the code, including External mode support code, free of doubles and floats. For more information, see "Code Generation Pane: 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 **Data exchange** section of the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box.



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, RSim, and Tornado 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 25-8. For more information on specifying a TCP/IP or serial transport layer for a custom target, see "Using the TCP/IP Implementation" on page 17-84 or "Using the Serial Implementation" on page 17-87.

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`, `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 17-83.

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)

External Mode Action	Toolbar Control	Menu Control	External Mode Control Panel Button
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 environment (does not halt running real-time code)	**Disconnect from Target** button	**Simulation** > **Disconnect from Target**	**Disconnect**
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 17-62.

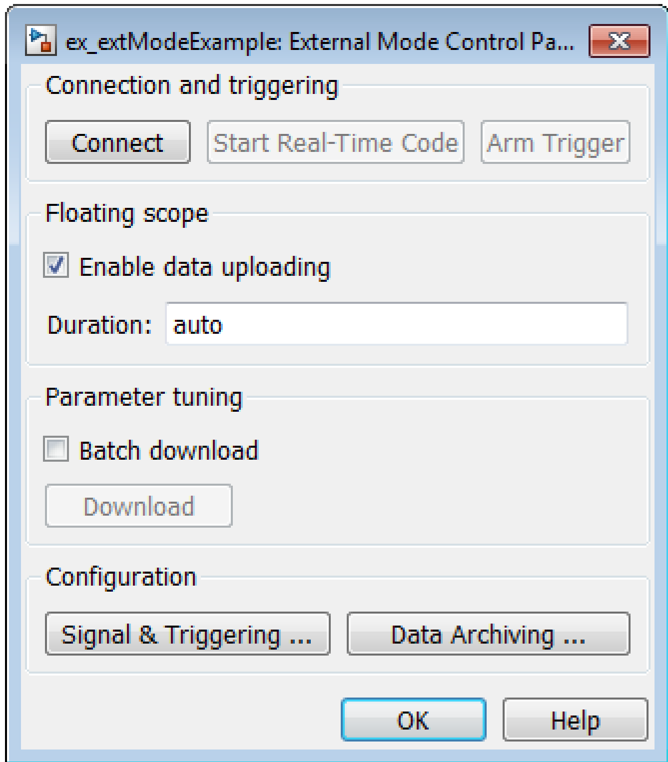
Control External Mode Operations

The External Mode Control Panel provides centralized control of External mode operations, including:

- Host/target connection and disconnection, target program start/stop functions, and enabling External mode.
- Arming and disarming the data upload trigger.
- External mode communication configuration.
- Uploading data to Floating Scopes.
- Timing of parameter downloads.
- Selection of signals from the target program to be viewed and monitored on the host.

- Configuration of data archiving.

To open the External Mode Control Panel dialog box, in the model window, select **Code** > **External Mode Control Panel**.



The following sections describe the operations supported by the External Mode Control Panel.

Connecting, Starting, and Stopping

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 17-66).

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 if it is not already set.

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.

Uploading Data to Floating Scopes

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.
- Support manual triggering only.

The behavior of wired scopes is not restricted in these ways.

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 or not to arm the trigger when connecting the floating scopes. When already connected, the option acts as a toggle button to arm or cancel the trigger.
- **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 (by default, 1000 base rate steps) to be used.

Parameter Downloading

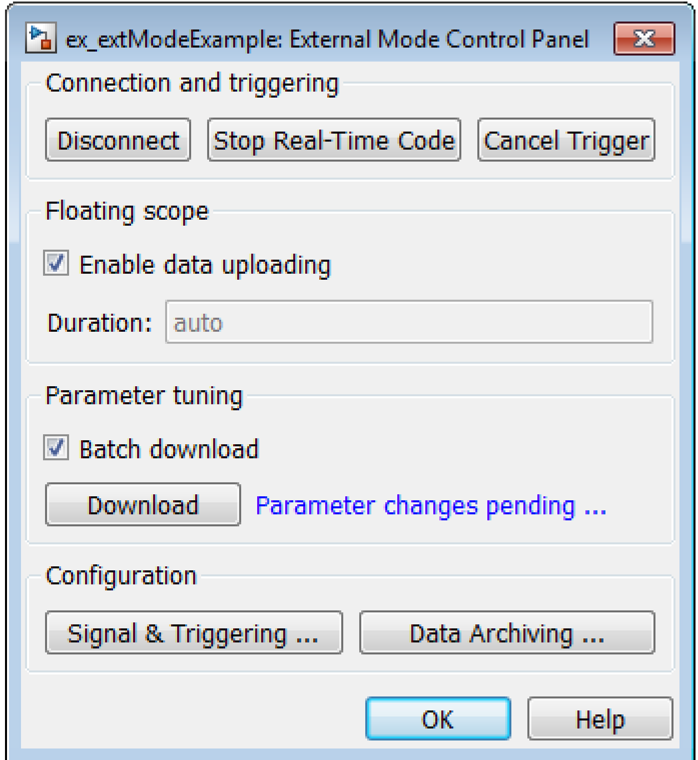
The **Batch download** option on the External Mode Control Panel enables or disables batch parameter changes.

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.

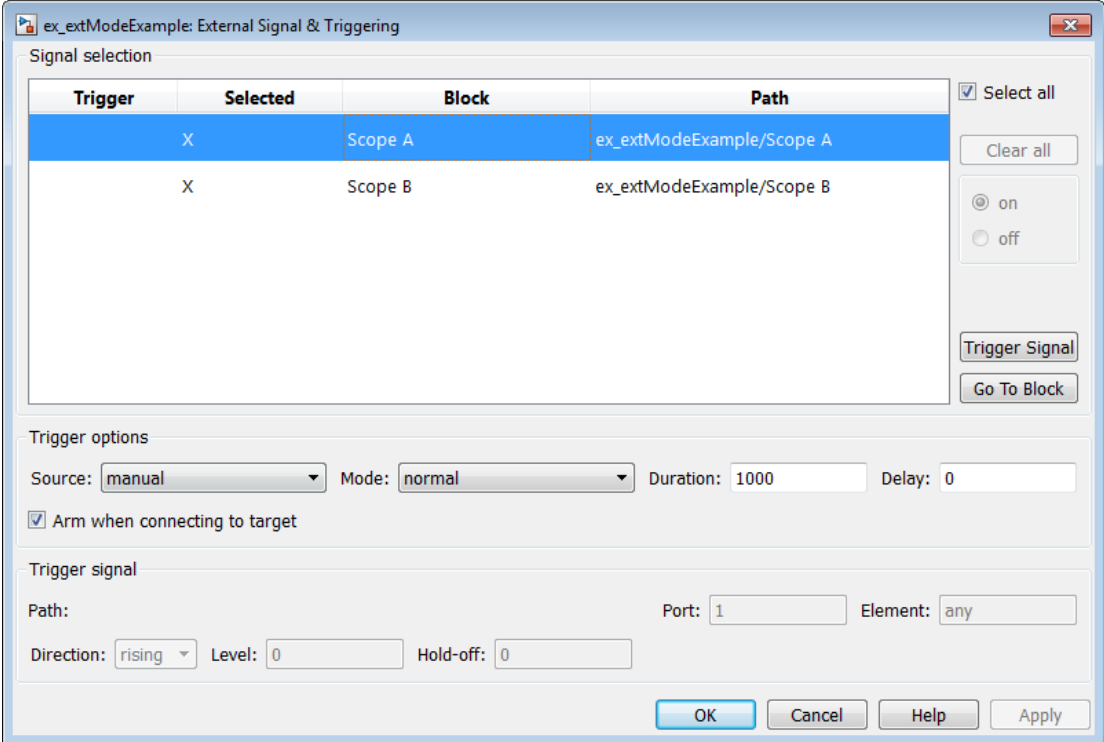
When parameter changes are awaiting batch download, the External Mode Control Panel displays the message `Parameter changes pending...` to the right of the **Download** button. This message remains visible until the Simulink engine receives notification from the target that the new parameters have been 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.



External Signal Uploading and Triggering

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 17-78.

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 Operation

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 will be selected and the

trigger will be 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
- `Arm when connecting to target`: on

Signal Selection

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.

Trigger Options

The **Trigger options** section in the External Signal & Triggering dialog contains options that control when and how signal data is collected (uploaded) from the target system. These options are:

- **Source**: `manual` or `signal`. Selecting `manual` directs External mode to use the **Arm trigger** button on the External Mode Control Panel as the trigger to start logging data. When you click **Arm trigger**, data logging begins.

  Selecting `signal` directs External mode to use a trigger signal as the trigger to start logging data. When a selected trigger signal satisfies trigger conditions specified in the **Trigger signal** section (that is, the signal crosses the trigger level in the specified direction), a *trigger event* occurs. If the trigger is *armed*, External mode monitors for the occurrence of a trigger event. When a trigger event occurs, data logging begins.
- **Mode**: `normal` or `one-shot`. In `normal` mode, External mode automatically rearms the trigger after each trigger event. 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 "Data Archiving" on page 17-75.

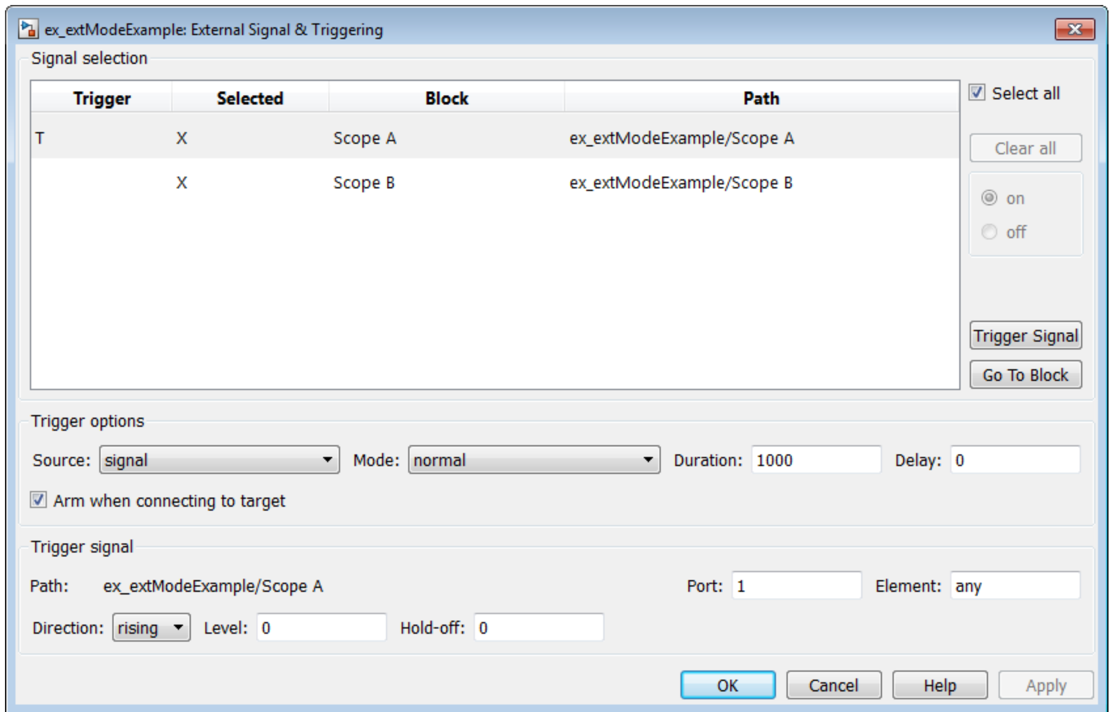
- **Duration**: The number of base rate steps for which External mode logs data after a trigger event (default is 1000). For example, if the fastest rate in the model is 1 second:
  - If a signal sampled at 1 Hz is logged for 10 base rate steps, External mode collects 10 samples.
  - If a signal sampled at 2 Hz is logged for 10 base rate steps, External mode collects 20 samples.
- **Delay**: The delay represents the amount of time that elapses between a trigger occurrence 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 is collected and uploaded.
- **Arm when connecting to target**: If you select this option, External mode arms the trigger automatically when the Simulink engine connects to the target. If the trigger source is `manual`, data uploading begins immediately. If the trigger mode is `signal`, monitoring of the trigger signal begins immediately, and data uploading begins upon the occurrence of a trigger event.

  If you clear **Arm when connecting to target**, you must manually arm the trigger by clicking the **Arm trigger** button on the External Mode Control Panel.

Trigger Signal Selection

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.

Setting 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.

Data Archiving

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 **Save 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

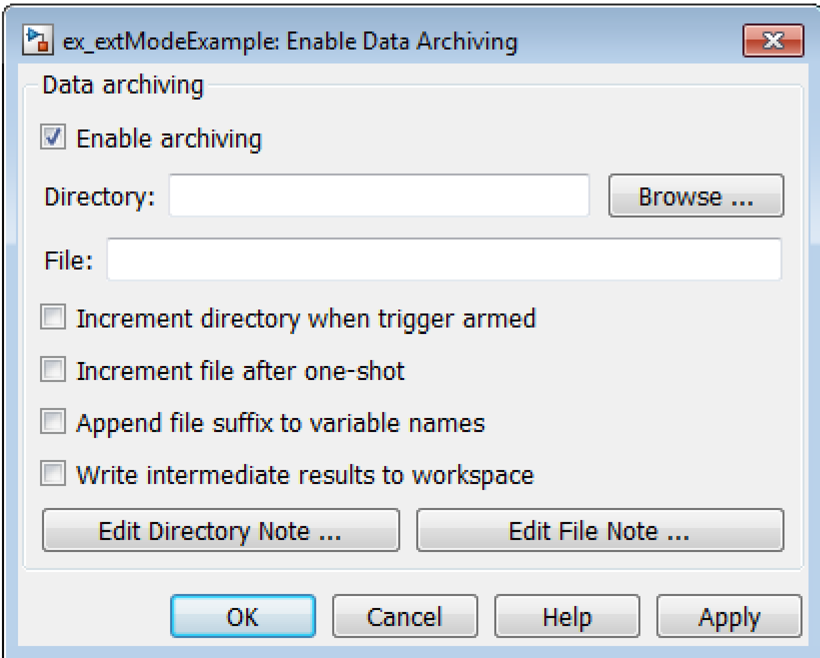
enables a variable named with the **Variable name** parameter to be saved to a MAT-file.

Note: If you do not select the Scope block option **Save data to workspace**, the MAT-files for data logging will be created, but they will be 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 17-78
- “Signal Viewing Subsystems” on page 17-78
- “Supported Blocks for Data Archiving” on page 17-80

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 Scope, Time Scope, and Vector Scope blocks from the DSP System Toolbox product
- Blocks from the Gauges Blockset 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/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 17-78 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.

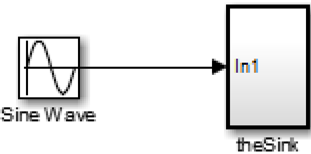
   For more information on atomic subsystems, see "Code Generation of Subsystems".
2. Use the following `set_param` command to turn the SimViewingDevice property on,

   ```
   set_param('blockname', 'SimViewingDevice','on')
   ```

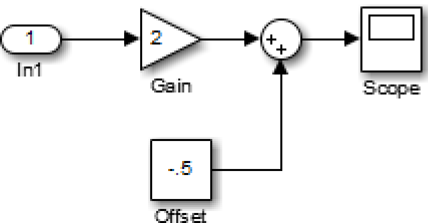
   where `'blockname'` is the name of the subsystem.
3. Make sure the subsystem meets the following requirements:

   - 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 "Data Archiving" on page 17-75.

External Mode Communication

- "About External Mode Communication" on page 17-80
- "Download Mechanism" on page 17-81
- "Inlined and Tunable Parameters" on page 17-82

About External Mode Communication

This section describes how the Simulink engine and a target program communicate, and how and when they transmit parameter updates and signal data to each other.

Depending on the setting of the **Inline parameters** option when the target program is generated, there are differences in the way parameter updates are handled. "Download Mechanism" on page 17-81 describes the operation of External mode communication with **Inline parameters** off. "Inlined and Tunable Parameters" on page 17-82 describes the operation of External mode with **Inline parameters** on.

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

mexFunction

Client IPC Code

External Interface

MEX-file (e.g., ext_comm)

External Program Process

External Program

Server IPC Code

ext_svr.

Interprocess Communication Channel Transport Layer

External Mode Architecture

Inlined and Tunable Parameters

By default, parameters (except those listed in “External Mode Limitations” on page 17-97) in an External mode program are tunable; that is, you can change them by using the download mechanism described in this section.

If you select the **Inline parameters** option (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 the Model Parameter Configuration dialog box to remove individual parameters from inlining and declare them to be tunable. In addition, the Model Parameter Configuration dialog box offers you options for controlling how parameters are represented in the generated code.

For more information on tunable parameters, see “Parameters” on page 8-11.

Automatic Parameter Uploading on Host/Target Connection

Each time the Simulink engine connects to a target program that was generated with **Inline parameters** on, 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 **Inline parameters** on. “Download Mechanism” on page 17-81 describes the operation of External mode communication with **Inline parameters** off.

Choose Communication Protocol for Client and Server

- “Introduction” on page 17-84
- “Using the TCP/IP Implementation” on page 17-84
- “Using the Serial Implementation” on page 17-87
- “Run the External Program” on page 17-89
- “Implement an External Mode Protocol Layer” on page 17-91

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 17-65 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 25-8.

To use Simulink External mode over TCP/IP, you must

- 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 **Host/Target interface** 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 the transport will be registered (for example, `'mytarget.tlc'`)

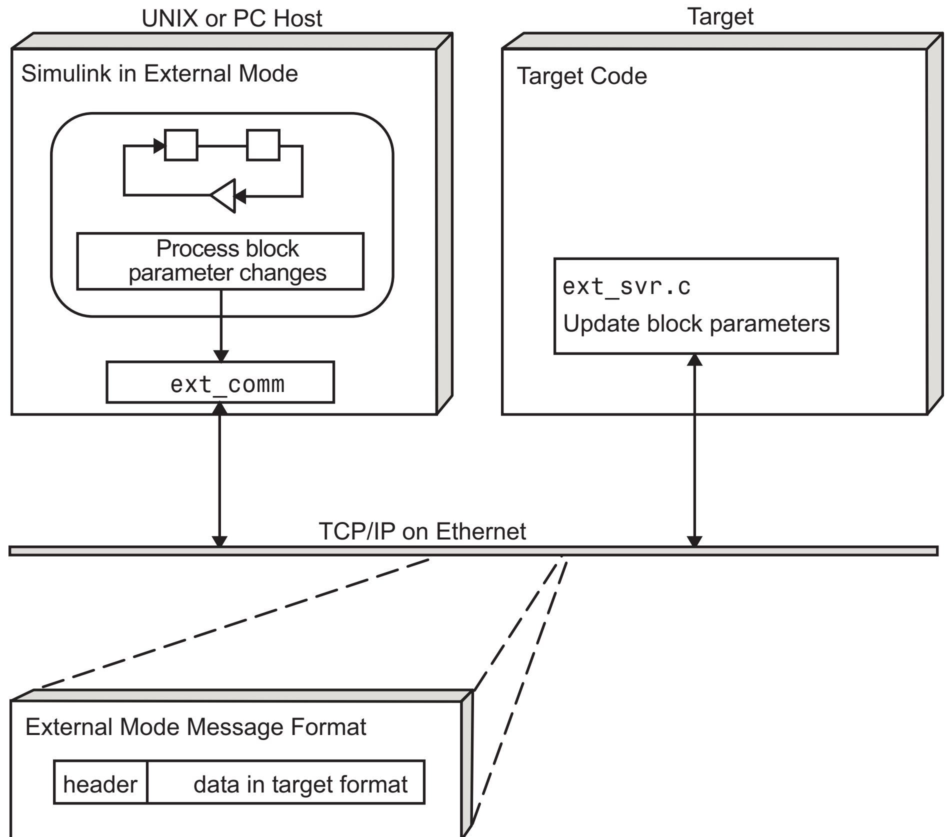
  - `transport` is the transport name to display in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box (for example, `'tcpip'`)
  - `mexfile` is the name of the transport's associated external interface MEX-file (for example, `'ext_comm'`)

  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. The name can be
  - String delimited by single quotes, such as `'myPuter'`
  - IP address delimited by single quotes, such as `'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 17-89 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 **Host/Target interface** 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 the transport will be registered (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 string. 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, 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 17-90.) 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 will be unable to 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 (`!`) 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 string. 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 25-8.

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`. (For more information on using `get_param` and `set_param` to tune model parameters, see “Tune Parameters” on page 8-34.)

The following sequence of model simulation commands assumes 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')
   ```

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 the `ExtMode` table entries in “Parameter Command-Line Information Summary”.

External Mode Command-Line Parameters

Parameter and Values	Dialog Box Equivalent	Description
`ExtModeAddSuffixToVar` **`off`**, `on`	Enable Data Archiving: **Append file suffix to variable names** check box	Increment variable names for each incremented filename.
`ExtModeArchiveDirName` *`string`*	Enable Data Archiving: **Directory** text field	Save data in specified folder.
`ExtModeArchiveFileName` *`string`*	Enable Data Archiving: **File** text field	Save data in specified file.
`ExtModeArchiveMode` *`string`* - **`off`**, `on`	Enable Data Archiving: **Enable archiving** check box	Activate automated data archiving features.
`ExtModeArmWhenConnect` `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.
`ExtModeAutoIncOneShot` **`off`**, `on`	Enable Data Archiving: **Increment file after one-shot** check box	Save new data buffers in incremental files.
`ExtModeAutoUpdateStatusClock` (Microsoft Windows platforms only) `off`, **`on`**	Not available	Continuously upload and display target time on the model window status bar.
`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` *`string`*	Not available	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.

Parameter and Values	Dialog Box Equivalent	Description
`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.
`ExtModeLogAll` `off`, **`on`**	External Signal & Triggering: **Select all** check box	Upload available signals from the target to the host.
`ExtModeLogCtrlPanelDlg` *`string`*	Not available	Return a handle to the External Mode Control Panel dialog box or –1 if the dialog box does not exist.
`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.

Parameter and Values	Dialog Box Equivalent	Description
`ExtModeTrigDirection` *string* - **`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` *string* - *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.
`ExtModeTrigElement` *string* - *integer*, **`any`**, `last`	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` *integer* (**0**)	External Signal & Triggering: **Hold-off** text field	Specify the base rate steps between when a trigger event terminates and the trigger is rearmed.
`ExtModeTrigLevel` *integer* (**0**)	External Signal & Triggering: **Level** text field	Specify the threshold value the trigger signal must cross to fire the trigger.
`ExtModeTrigMode` *string* - **`normal`**, `oneshot`	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` *string* - *integer* (**1**), `last`	External Signal & Triggering: **Port** text field	Specify the input port of the specified trigger block for which elements can cause the trigger to fire.

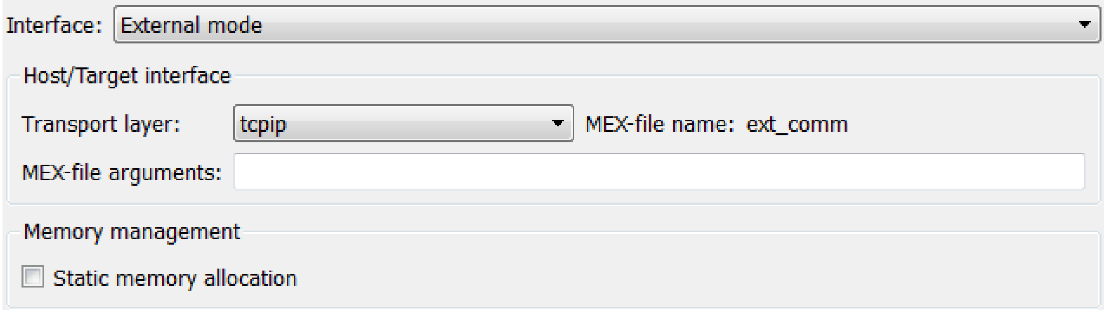
Parameter and Values	Dialog Box Equivalent	Description
`ExtModeTrigType` *`string`* - **`manual`**, `signal`	External Signal & Triggering: **Source** menu	Specify whether to start logging data when the trigger is armed or when a specified trigger signal satisfies trigger conditions.
`ExtModeUploadStatus` *`string`* - `inactive`, `armed`, `uploading`	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.

Generate External Mode and C API Data Interfaces

The External mode and C API data interfaces for model code are not mutually exclusive. You can generate code for your model with both the External mode and C API interfaces enabled. This allows C API data structures to be accessed from custom code during an External mode simulation.

To configure your model so that External mode and the C API can be used together, you must configure at least one of the data interfaces at the MATLAB command line. The following example configures External mode in the Configuration Parameters dialog box and configures the C API at the command line.

1 On the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box, select `External mode` for the **Interface** parameter. Inspect the settings for the other External mode related controls and make your adjustments. Click **Apply**.



2 In the MATLAB Command Window, with the model open, specify one or more of the following commands, depending on which data items you want to be able to access with the C API:

```
>> set_param(gcs,'RTWCAPIParams','on')
>> set_param(gcs,'RTWCAPISignals','on')
>> set_param(gcs,'RTWCAPIStates','on')
>> set_param(gcs,'RTWCAPIRootIO','on')
```

Note: If you select both the External mode and C API data interfaces, the Configuration Parameters dialog box (in subsequent opens) displays only the C API options. You can display the current External mode parameter settings using `get_param` commands.

---

3 Click **Generate Code** or **Build**. The build process generates code to support both C API data access (including the *`model`*`_capi` source and header files) and External mode data access.

External Mode Limitations

- "Limitation on Changing Parameters" on page 17-98
- "Limitation on Mixing 32-bit and 64-bit Architectures" on page 17-98
- "Limitation on Uploading Data" on page 17-99
- "Limitation on Uploading Variable-Size Signals" on page 17-99
- "Limitation on Archiving Data" on page 17-99
- "Limitation on Scopes in Referenced Models" on page 17-99

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 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.

Logging

- “Log Data for Analysis” on page 17-100
- “About Logging to MAT-Files” on page 17-106
- “Configure State, Time, and Output Logging” on page 17-107

- “Log Data with Scope and To Workspace Blocks” on page 17-109
- “Log Data with To File Blocks” on page 17-109
- “Data Logging Differences Between Single- and Multitasking” on page 17-109

Log Data for Analysis

- “About Logging Data” on page 17-100
- “Data Logging During Simulation” on page 17-101
- “Data Logging from Generated Code” on page 17-104

About Logging Data

Simulink Coder MAT-file data logging facility 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`, where `model` is the name of your model. In this example, 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 17-19 for instructions on setting up a copy of `slexAircraftExample` as`myAircraftExample` in a working folder if you have not done so already.

Note: If you want the code generator to produce code that includes support for data logging during execution, use the ASCII-7 character set when naming blocks. Otherwise, the block names in block paths included in the log file are not readable and do not compile. For more information, see “International Character Support” on page 17-5.

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 the **MAT-file variable name modifier** parameter on the **Code Generation** > **Interface** pane.

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: Signal logging is not supported when MAT-file logging is enabled.
When your model code executes, the signal logging variable 'rt_logsout' will
not be saved to the MAT-file.
```

To avoid this warning, clear the **Data Import/Export** > **Signal logging** check box.

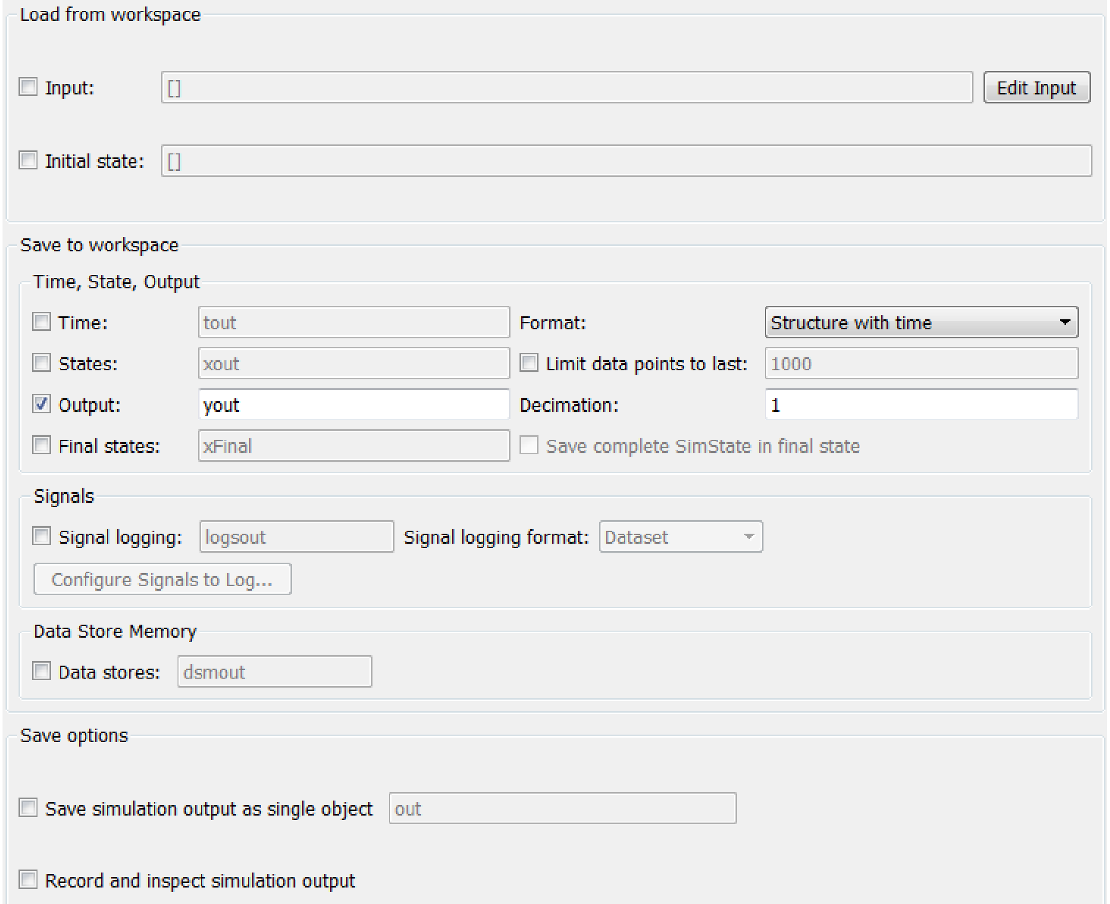
In this example, you modify the `myAircraftExample` model so that the generated program saves the simulation time and system outputs to the file `myAircraftExample.mat`. Then you load the data into the base workspace and plot simulation time against one of the outputs. The `myAircraftExample` model should be configured as described in “Build a Generic Real-Time Program” on page 17-19.

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 outport 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** 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`.

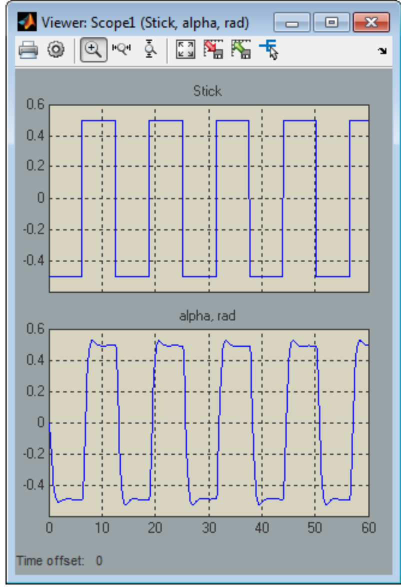
7 If other options are enabled, clear them. The figure below shows how the dialog box should appear.



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
```

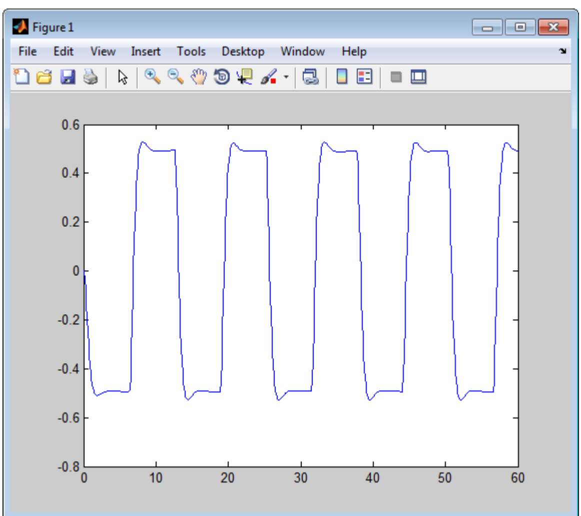
Simulink displays:

```
  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.

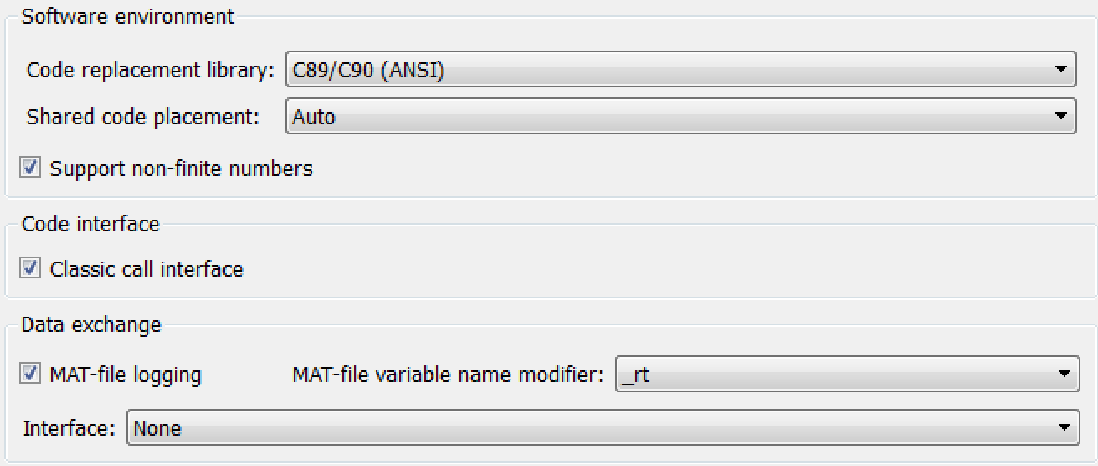


Data Logging from Generated Code

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 from the Configuration Parameters dialog box:

1. Open the Configuration Parameters dialog box by selecting **Simulation > Model Configuration Parameters** in the model window.
2. Select the **Code Generation > Interface** pane.
3. Set **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. The pane should look like this:



4. Click **Apply** and **OK** to register your changes and close the dialog box.
5. Save the model.
6. Build an executable, by clicking the **Build Model** button in the Simulink Editor toolbar.
7. When the build concludes, run the executable with the command:

```
!myAircraftExample
```

8. The program now produces two message lines, indicating that the MAT-file has been written.

```
** starting the model **
** created myAircraftExample.mat **
```

9. 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*
```

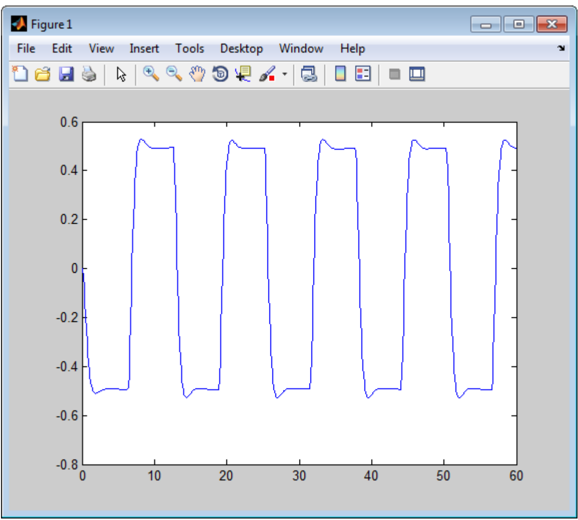
Simulink displays:

```
  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.

10 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.

About Logging to MAT-Files

Multiple techniques are available by which a program generated by the Simulink Coder software can save data to a MAT-file for analysis. See also "Log Data for Analysis" on page 17-100 for a data logging tutorial.

Note: Data logging is available only for targets that have access to a file system. In addition, only the RSim target executables are capable of accessing MATLAB workspace data.

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 17-107
- “Override Default MAT-File Name or Buffer Size” on page 17-108

Override Default MAT-File Variable Names

By default, the code generation software prefixes the string `rt_` to the variable names for system outputs, states, and simulation time to form MAT-file variable names. To change this prefix for a GRT or ERT-based model,

1. Open the Configuration Parameters dialog box and select the **Code Generation** > **Interface** pane.

2 For the **MAT-file variable name modifier** parameter, select a prefix (`rt_`), a suffix (`_rt`), or no modifier (`none`). Other targets may or may not have this option.

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	*model*.mat	-DSAVEFILE=*filename*
Size of data logging buffer	1024 bytes	-DDEFAULT_BUFFER_SIZE=*n*

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	$(cflags) $(CVARSFLAG) $(CFLAGS_ADDITIONAL) -DSAVEFILE=myCodeLog.mat /Od /Oy-

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 Save 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" for more details on the differences between single-tasking and multitasking data logging.

Note: The rapid simulation target (RSim) provides enhanced logging options. See "Rapid Simulations" for more information.

Parameter Tuning

- "Tunable Parameter Storage" on page 17-14
- "Tunable Parameter Storage Classes" on page 17-15
- "Declare Tunable Parameters" on page 17-17
- "Tunable Expressions" on page 17-21
- "Linear Block Parameter Tunability" on page 17-25
- "Tunable Workspace Parameter Data Type Considerations" on page 17-32
- "Tune Parameters from the Command Line" on page 17-34
- "Interfaces for Tuning Parameters" on page 17-35

Tunable Parameter Storage

A *tunable* parameter is a block parameter whose value can be changed at run-time. A tunable parameter is inherently noninlined. Consequently, when **Inlined parameters** is off, all parameters are members of `model_P`, and thus are tunable. A *tunable expression* is an expression that contains one or more tunable parameters.

When you declare a parameter tunable, you control whether or not the parameter is stored within `model_P`. You also control the symbolic name of the parameter in the generated code.

When you declare a parameter tunable, you specify

- The *storage class* of the parameter.

  The storage class property of a parameter specifies how the Simulink Coder product declares the parameter in generated code.

  The term "storage class," as used in the Simulink Coder product, is not synonymous with the term *storage class specifier*, as used in the C language.
- A *storage type qualifier*, such as `const` or `volatile`. This is simply a string that is included in the variable declaration.
- (Implicitly) the symbolic name of the variable or field in which the parameter is stored. The Simulink Coder product derives variable and field names from the names of tunable parameters.

The Simulink Coder product generates a variable or `struct` storage declaration for each tunable parameter. Your choice of storage class controls whether the parameter is declared as a member of *`model`*`_P` or as a separate global variable.

You can use the generated storage declaration to make the variable visible to external legacy code. You can also make variables declared in your code visible to the generated code. You are responsible for linking your code to generated code modules.

You can use tunable parameters or expressions in your root model and in masked or unmasked subsystems, subject to certain restrictions. (See "Tunable Expressions" on page 8-21.)

Override Inlined Parameters for Tuning

When the **Inline parameters** option is selected, you can use the Model Parameter Configuration dialog box to remove individual parameters from inlining and declare them to be tunable. This allows you to improve overall efficiency by inlining most parameters, while at the same time retaining the flexibility of run-time tuning for selected parameters. Another way you can achieve the same result is by using Simulink data objects; see "Parameters" on page 8-11 for specific details.

The mechanics of declaring tunable parameters are discussed in "Declare Tunable Parameters" on page 8-17.

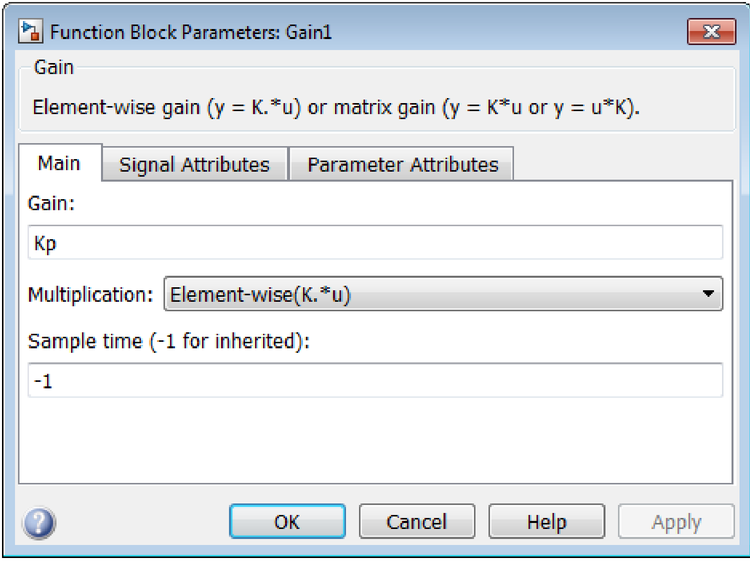
Tunable Parameter Storage Classes

The Simulink Coder product defines four storage classes for tunable parameters. You must declare a tunable parameter to have one of the following storage classes:

- `SimulinkGlobal (Auto)`: This is the default storage class. The Simulink Coder product stores the parameter as a member of *`model`*`_P`. Each member of *`model`*`_P` is initialized to the value of the corresponding workspace variable at code generation time.
- `ExportedGlobal`: The generated code instantiates and initializes the parameter and *`model`*`.h` exports it as a global variable. An exported global variable is independent of the *`model`*`_P` data structure. Each exported global variable is initialized to the value of the corresponding workspace variable at code generation time.
- `ImportedExtern`: *`model`*`_private.h` declares the parameter as an `extern` variable. Your code must supply the variable definition and initializer.
- `ImportedExternPointer`: *`model`*`_private.h` declares the variable as an `extern` pointer. Your code must supply the pointer variable definition and initializer, if any.

The generated code for *`model`*`.h` includes *`model`*`_private.h` to make the `extern` declarations available to subsystem files.

As an example of how the storage class declaration affects the code generated for a parameter, consider the next figure.



The workspace variable `Kp` sets the gain of the `Gain1` block. Assume that the value of `Kp` is 3.14. The following table shows the variable declarations and the code generated for

the gain block when Kp is declared as a tunable parameter. An example is shown for each storage class.

Note The Simulink Coder product uses column-major ordering for two-dimensional signal and parameter data. When interfacing your hand-written code to such signals or parameters by using ExportedGlobal, ImportedExtern, or ImportedExternPointer declarations, make sure that your code observes this ordering convention.

The symbolic name Kp is preserved in the variable and field names in the generated code.

Storage Class	Generated Variable Declaration and Code

SimulinkGlobal (Auto)	typedef struct _Parameters_tunable_sin Parameters_tunable_sin;
	
	struct _Parameters_tunable_sin {
	  real_T Kp;
	};
	
	Parameters_tunable_sin tunable_sin_P = {
	  3.14
	};
	.
	.
	tunable_sin_Y.Out1 = rtb_u * tunable_sin_P.Kp;

ExportedGlobal	real_T Kp = 3.14;
	.
	.
	tunable_sin_Y.Out1 = rtb_u * Kp;

ImportedExtern	extern real_T Kp; . . tunable_sin_Y.Out1 = rtb_u * Kp;
ImportedExtern Pointer	extern real_T *Kp; . . tunable_sin_Y.Out1 = rtb_u * (*Kp);

Declare Tunable Parameters

- "Declare Workspace Variables as Tunable Parameters" on page 17-17
- "Declare New Tunable Parameters" on page 17-18
- "Declare Tunable Parameters Using Configuration Dialog" on page 17-18
- "Select Workspace Variables" on page 17-19
- "Create New Tunable Parameters" on page 17-20
- "Set Tunable Parameter Properties" on page 17-20
- "Remove Unused Tunable Parameters" on page 17-21

Declare Workspace Variables as Tunable Parameters

To declare tunable parameters,

1. Open the Model Parameter Configuration dialog box.
2. In the **Source list** pane, select one or more variables.
3. Click **Add to table** . The variables then appear as tunable parameters in the **Global (tunable) parameters** pane.
4. Select a parameter in the **Global (tunable) parameters** pane.
5. Select a storage class from the **Storage class** menu.
6. Optionally, select (or enter) a storage type qualifier, such as `const` or `volatile` for the parameter.
7. Click **Apply**, or click **OK** to apply changes and close the dialog box.

Declare New Tunable Parameters

To declare tunable parameters,

1. Open the Model Parameter Configuration dialog box.
2. In the **Global (tunable) parameters** pane, click **New**.
3. Specify a name for the parameter.
4. Select a storage class from the **Storage class** menu.
5. Optionally, select (or enter) a storage type qualifier, such as `const` or `volatile` for the parameter.
6. Click **Apply**, or click **OK** to apply changes and close the dialog box.

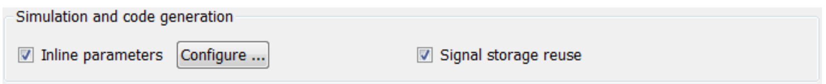
Declare Tunable Parameters Using Configuration Dialog

The Model Parameter Configuration dialog box lets you select base workspace variables and declare them to be tunable parameters in the current model. Using controls in the

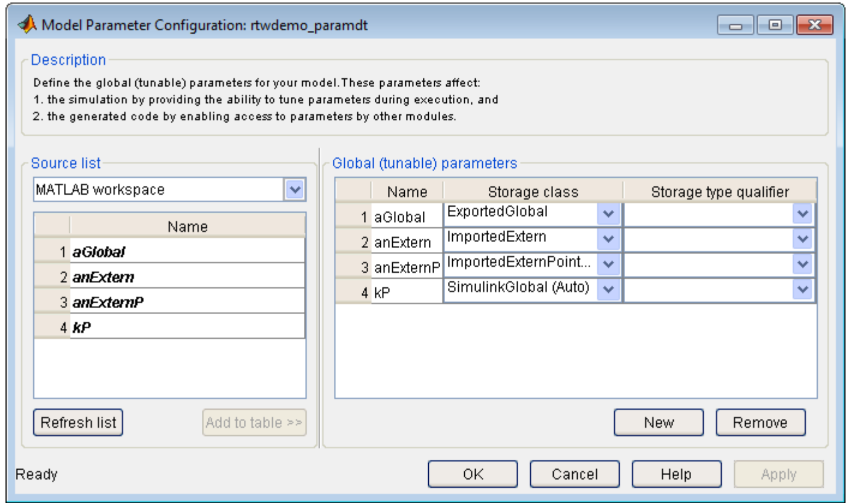
dialog box, you move variables from a source list to a global (tunable) parameter list for a model.

To open the dialog box,

1. Select the **Inline parameters** check box on the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box. This activates a **Configure** button, as shown below.



2. Click **Configure** to open the Model Parameter Configuration dialog box.



Note The Model Parameter Configuration dialog box cannot tune parameters within referenced models. See "Parameterize Model References" for tuning techniques that work with referenced models.

Select Workspace Variables

The **Source list** pane displays a menu and a scrolling table of numerical workspace variables. To select workspace variables,

1 From the menu, select the source of variables you want listed.

To List...	Choose...
Variables in the MATLAB workspace that have numeric values	`MATLAB workspace`
Only variables in the MATLAB workspace that have numeric values and are referenced by the model	`Referenced workspace variables`

A list of workspace variables appear in the **Source List** pane.

2 Select one or more variables from the source list. This enables the **Add to table** button.

3 Click **Add to table** to add the selected variables to the tunable parameters list in the **Global (tunable) parameters** pane. In the **Source list**, the names of variables added to the tunable parameters list are displayed in bold type (see the preceding figure).

Note: If you selected a variable with a name that matches a block parameter that is not tunable and you click **Add to table** , a warning appears during simulation and code generation.

To update the list of variables to reflect the current state of the workspace, click **Refresh list**. For example, you might use **Refresh list** if you define or remove variables in the workspace while the Model Parameter Configuration dialog box is open.

Create New Tunable Parameters

To create a new tunable parameter,

1 In the **Global (tunable) parameters** pane, click **New**.

2 In the **Name** field, enter a name for the parameter.

If you enter a name that matches the name of a workspace variable in the **Source list** pane, that variable is declared tunable and appears in italics in the **Source list**.

3 Click **Apply**.

The model does not need to be using a parameter before you create it. You can add references to the parameter later.

Note If you edit the name of an existing variable in the list, you actually create a new tunable variable with the new name. The previous variable is removed from the list and loses its tunability (that is, it is inlined).

Set Tunable Parameter Properties

To set the properties of tunable parameters listed in the **Global (tunable) parameters** pane, select a parameter and then specify a storage class and, optionally, a storage type qualifier.

Property	Description
Storage class	Select one of the following to be used for code generation: • `SimulinkGlobal (Auto)` • `ExportedGlobal` • `ImportedExtern` • `ImportedExternPointer` See “Tunable Parameter Storage Classes” on page 8-15 for definitions.
Storage type qualifier	For variables with a storage class *except* `SimulinkGlobal (Auto)`, you can add a qualifier (such as `const` or `volatile`) to the generated storage declaration. To do so, you can select a predefined qualifier from the list or add qualifiers not in the list. The code generator does not check the storage type qualifier for validity, and includes the qualifier string in the generated code without checking syntax .

Remove Unused Tunable Parameters

To remove unused tunable parameters from the table in the **Global (tunable) parameters** pane, click **Remove**. All removed variables are inlined if the **Inlined parameters** option is enabled.

Tunable Expressions

- “Tunable Expressions in Masked Subsystems” on page 17-21

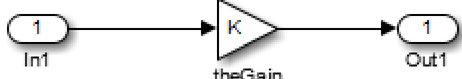
- "Tunable Expression Limitations" on page 17-23

The Simulink Coder product supports the use of tunable variables in expressions. An expression that contains one or more tunable parameters is called a *tunable expression*.

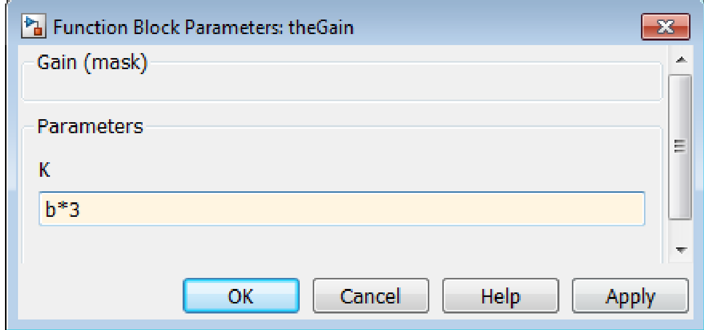
Tunable Expressions in Masked Subsystems

Tunable expressions are allowed in masked subsystems. You can use tunable parameter names or tunable expressions in a masked subsystem dialog box. When referenced in lower-level subsystems, such parameters remain tunable.

As an example, consider the masked subsystem in the next figure. The masked variable k sets the gain parameter of theGain.



Suppose that the base workspace variable b is declared tunable with SimulinkGlobal (Auto) storage class. The next figure shows the tunable expression b*3 in the subsystem's mask dialog box.



Tunable Expression in Subsystem Mask Dialog Box

The Simulink Coder product produces the following output computation for theGain. The variable b is represented as a member of the global parameters structure, *model*_P. (For clarity in showing the individual Gain block computation, expression folding is off in this example.)

```
/* Gain: '<S1>/theGain' */
```

```
rtb_theGain_C = rtb_SineWave_n * ((subsys_mask_P.b * 3.0));

/* Outport: '<Root>/Out1' */
subsys_mask_Y.Out1 = rtb_theGain_C;
```

As this example shows, for GRT targets, the parameter structure is mangled to create the structure identifier `model_P` (subject to the identifier length constraint). This is done to avoid namespace clashes in combining code from multiple models using model reference. ERT-based targets provide ways to customize identifier names.

When expression folding is on, the above code condenses to

```
/* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<S1>/theGain'
   */
 subsys_mask_Y.Out1 = rtb_SineWave_n * ((subsys_mask_P.b * 3.0));
```

Expressions that include variables that were declared or modified in mask initialization code are *not* tunable.

As an example, consider the subsystem above, modified as follows:

- The mask initialization code is

  ```
  t = 3 * k;
  ```
- The parameter `k` of the `myGain` block is `4 + t`.
- Workspace variable `b = 2`. The expression `b * 3` is plugged into the mask dialog box as in the preceding figure.

Since the mask initialization code can run only once, `k` is evaluated at code generation time as

```
4 + (3 * (2 * 3) )
```

The Simulink Coder product inlines the result. Therefore, despite the fact that `b` was declared tunable, the code generator produces the following output computation for `theGain`. (For clarity in showing the individual Gain block computation, expression folding is off in this example.)

```
/* Gain Block: <S1>/theGain */
rtb_temp0 *= (22.0);
```

Tunable Expression Limitations

Currently, there are certain limitations on the use of tunable variables in expressions. When an unsupported expression is encountered during code generation a warning is

issued and the equivalent numeric value is generated in the code. The limitations on tunable expressions are

- Complex expressions are not supported, except where the expression is simply the name of a complex variable.
- The use of certain operators or functions in expressions containing tunable operands is restricted. Restrictions are applied to four categories of operators or functions, classified in the following 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`,
4	`single`, `int8`, `int16`, `int32`, `uint8`, `uint16`, `uint32`
5	`: .^ ^ [] {} . \ .\ ' .' ; ,`

The rules applying to each category are as follows:

- Category 1 is unrestricted. These operators can be used in tunable expressions with any combination of scalar or vector operands.
- Category 2 operators can be used in tunable expressions where at least one operand is a scalar. That is, scalar/scalar and scalar/matrix operand combinations are supported, but not matrix/matrix.
- Category 3 lists all functions that support tunable arguments. Tunable arguments passed to these functions retain their tunability. Tunable arguments passed to any other functions lose their tunability.
- Category 4 lists the casting functions that do not support tunable arguments. Tunable arguments passed to these functions lose their tunability.

Note: The Simulink Coder product casts values using MATLAB typecasting rules. The MATLAB typecasting rules are different from C code typecasting rules. For example, using the MATLAB typecasting rules, `int8(3.7)` returns the result `4`, while in C code `int8(3.7)` returns the result `3`. See "Data Type Conversion" for more information on MATLAB typecasting.

- Category 5 operators are not supported.

- Expressions that include variables that were declared or modified in mask initialization code are *not* tunable.
- The Fcn block does not support tunable expressions in code generation.
- Model workspace parameters can take on only the `Auto` storage class, and thus are not tunable. See “Parameterize Model References” for tuning techniques that work with referenced models.
- Non-double expressions are not supported.
- Blocks that access parameters only by address support the use of tunable parameters, if the parameter expression is a simple variable reference. When an operation such as a data type conversion or a math operation is applied, the Simulink Coder product creates a nontrivial expression that cannot be accessed by address, resulting in an error during the build process.

Linear Block Parameter Tunability

The following blocks have a `Realization` parameter that affects the tunability of their parameters:

- Transfer Fcn
- State-Space
- Discrete State-Space

The `Realization` parameter must be set by using the MATLAB `set_param` function, as in the following example.

```
set_param(gcb,'Realization','auto')
```

The following values are defined for the Realization parameter:

- `general`: The block's parameters are preserved in the generated code, permitting parameters to be tuned.
- `sparse`: The block's parameters are represented in the code by transformed values that increase the computational efficiency. Because of the transformation, the block's parameters are no longer tunable.
- `auto`: This setting is the default. A `general` realization is used if one or more of the block's parameters are tunable. Otherwise `sparse` is used.

Note To tune the parameter values of a block of one of the above types without restriction during an External mode simulation, you must set Realization to `general`.

Code Reuse for Subsystems with Mask Parameters

The Simulink Coder product can generate 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 any of the 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 Simulink Coder product will generate separate functions for A and B.

Tunable Workspace Parameter Data Type Considerations

If you are using tunable workspace parameters, you need to be aware of potential issues regarding data types. A workspace parameter is tunable when the following conditions exist:

- You select the **Inline parameters** option on the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box
- The parameter has a storage class other than `Auto`

When generating code for tunable workspace parameters, the Simulink Coder product checks and compares the data types used for a particular parameter in the workspace and in Block Parameter dialog boxes.

If...	The Simulink Coder Product...
The data types match	Uses that data type for the parameter in the generated code.

You do not explicitly specify a data type other than `double` in the workspace	Uses the data type specified by the block in the generated code. If multiple blocks share a parameter, they must all specify the same data type. If the data type varies between blocks, the product generates an error similar to the following:
	`Variable 'K' is used in incompatible ways`
	`in the dialog fields of the following:`
	`cs_params/Gain, cs_params/Gain1. The`
	`variable'value is being used both directly`
	`and after a transformation. Only one of`

If...	The Simulink Coder Product...
	these usages is permitted for a given variable.
You explicitly specify a data type other than `double` in the workspace	Uses the data type from the workspace for the parameter. The block typecasts the parameter to the block specific data type before using it.

Guidelines for Specifying Data Types

The following table provides guidelines on specifying data types for tunable workspace parameters.

If You Want to...	Then Specify Data Types in...
Minimize memory usage (`int8` instead of `single`)	The workspace explicitly
Avoid typecasting	Blocks only
Interface to legacy or custom code	The workspace explicitly
Use the same parameter for multiple blocks that specify different data types	The workspace explicitly

The Simulink Coder product enforces limitations on the use of data types other than `double` in the workspace, as explained in "Limitations on Data Type Specifications in Workspace" on page 8-33.

Limitations on Specifying Workspace Data Types Explicitly

When you explicitly specify a data type other than `double` in the workspace, blocks typecast the parameter to another data type. This is an issue for blocks that use pointer access for their parameters. Blocks cannot use pointer access if they need to typecast the parameter before using it (because of a data type mismatch). Another case in which this occurs is for workspace variables with bias or fractional slope. Two possible solutions to these problems are

- Remove the explicit data type specification in the workspace for parameters used in such blocks.
- Modify the block so that it uses the parameter with the same data type as specified in the workspace. For example, the Lookup Table block uses the data types of its input signal to determine the data type that it uses to access the `X-breakpoint` parameter. You can prevent the block from typecasting the run-time parameter by converting the

input signal to the data type used for `X-breakpoints` in the workspace. (Similarly, the output signal is used to determine the data types used to access the lookup table `Y` data.)

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 "Configure Parameter Objects for Code Generation" on page 8-36 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` (string) is a comma-separated list of variable names.
- `set_param(gcs, 'TunableVarsStorageClass', str)`

  The argument `str` (string) 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` (string) 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')
```

Other configuration parameters you can get and set are listed in “Parameter Reference”.

Interfaces for Tuning Parameters

The Simulink Coder product includes

- Support for developing a Target Language Compiler API for tuning parameters independent of External mode. See “Parameter Functions” in the Target Language Compiler documentation for information.
- A C application program interface (API) for tuning parameters independent of External mode. See “Data Interchange Using the C API” on page 17-125 for information.
- An interface for exporting ASAP2 files, which you customize to use parameter objects. For details, see “ASAP2 Data Measurement and Calibration” on page 17-158.

Data Interchange Using the C API

The C API allows you to write host-based or target-based code that interacts with signals, states, root-level inputs/outputs, and parameters in your target-based application code.

- “About Data Exchange and C API” on page 17-125
- “Generate C API Files” on page 17-127
- “Description of C API Files” on page 17-129
- “Use the C API in an Application” on page 17-145
- “C API Limitations” on page 17-157

About Data Exchange and C API

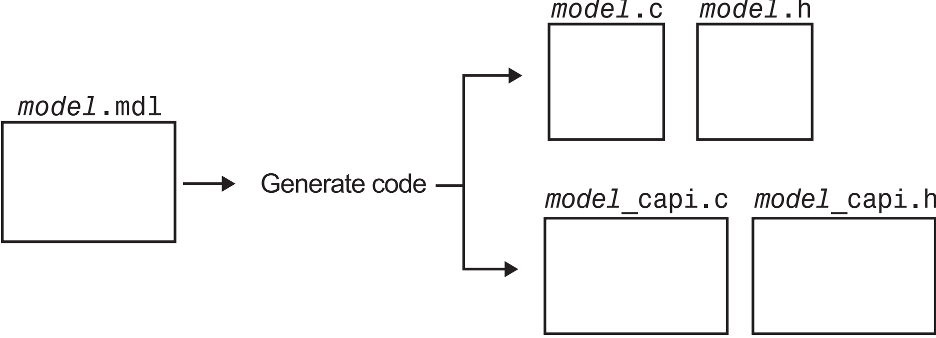
Some Simulink Coder applications must interact with signals, states, root-level inputs/outputs, or parameters in the generated code for a model. For example, calibration applications monitor and modify parameters. Signal monitoring or data logging

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.

Note: If you want the code generator to produce code that includes support for data logging during execution, use the ASCII-7 character set when naming blocks. Otherwise, the block names in block paths included in the log file are not readable and do not compile. For more information, see "International Character Support" on page 17-5.

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.

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

Generate C API Files

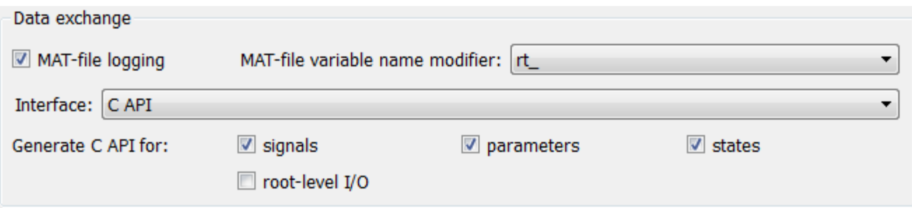
To generate C API files for your model:

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 17-127
   - "Select C API from the Command Line" on page 17-128
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 select **Simulation** > **Model Configuration Parameters** to launch the Configuration Parameters dialog box.
2. Go to the **Code Generation** > **Interface** pane and, in the **Data exchange** section, select `C API` as the value for the **Interface** parameter. The **Generate C API for: signals**, **Generate C API for: parameters**, **Generate C API for: states**, and **Generate C API for: root-level I/O** check boxes are displayed.



3. Select options:
   - If you want to generate C API code for global block output signals, select the **Generate C API for: signals** check box.
   - If you want to generate C API code for global block parameters, select the **Generate C API for: parameters** check box.
   - If you want to generate C API code for discrete and continuous states, select the **Generate C API for: states** check box.
   - If you want to generate C API code for root-level inputs and outputs, select the **Generate C API for: root-level I/O** check box.

If you select the four check boxes, support for accessing signals, parameters, states, and root-level I/O will appear in the C API generated code.

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','RTWCAPIStates','on')
```

To clear **Generate C API for: states**, enter:

```
set_param('modelname','RTWCAPIStates','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')
```

Generate C API and ASAP2 Data Interfaces

The C API and ASAP2 data interfaces for model code are not mutually exclusive. You can generate code for your model with both the C API and ASAP2 interfaces enabled. For

information on how to configure your model so that the build process generates files for both the C API and ASAP2 interfaces, see “Generate ASAP2 and C API Data Interfaces” on page 17-170.

Generate C API and External Mode Data Interfaces

The C API and External mode data interfaces for model code are not mutually exclusive. You can generate code for your model with both the C API and External mode interfaces enabled. This allows C API data structures to be accessed from custom code during an External mode simulation. For information on how to configure your model so that the C API and External mode can be used together, see “Generate External Mode and C API Data Interfaces” on page 17-96.

Description of C API Files

- “About C API Files” on page 17-129
- “Structure Arrays Generated in C API Files” on page 17-132
- “Generate Example C API Files” on page 17-133
- “C API Signals” on page 17-136
- “C API States” on page 17-139
- “C API Root-Level Inputs and Outputs” on page 17-140
- “C API Parameters” on page 17-141
- “Map C API Data Structures to rtModel” on page 17-143

About C API Files

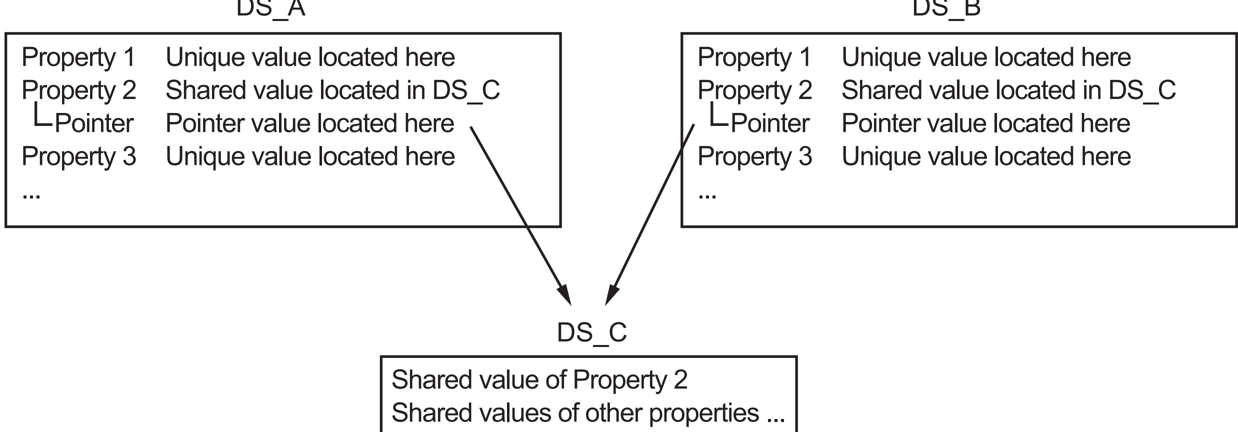
The *model*`_capi.c` (or `.cpp`) file provides external applications with a consistent interface to model data. Depending on your configuration settings, the data could be a signal, state, root-level input or output, or parameter. In this document, the term *data item* refers to either a signal, a state, 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

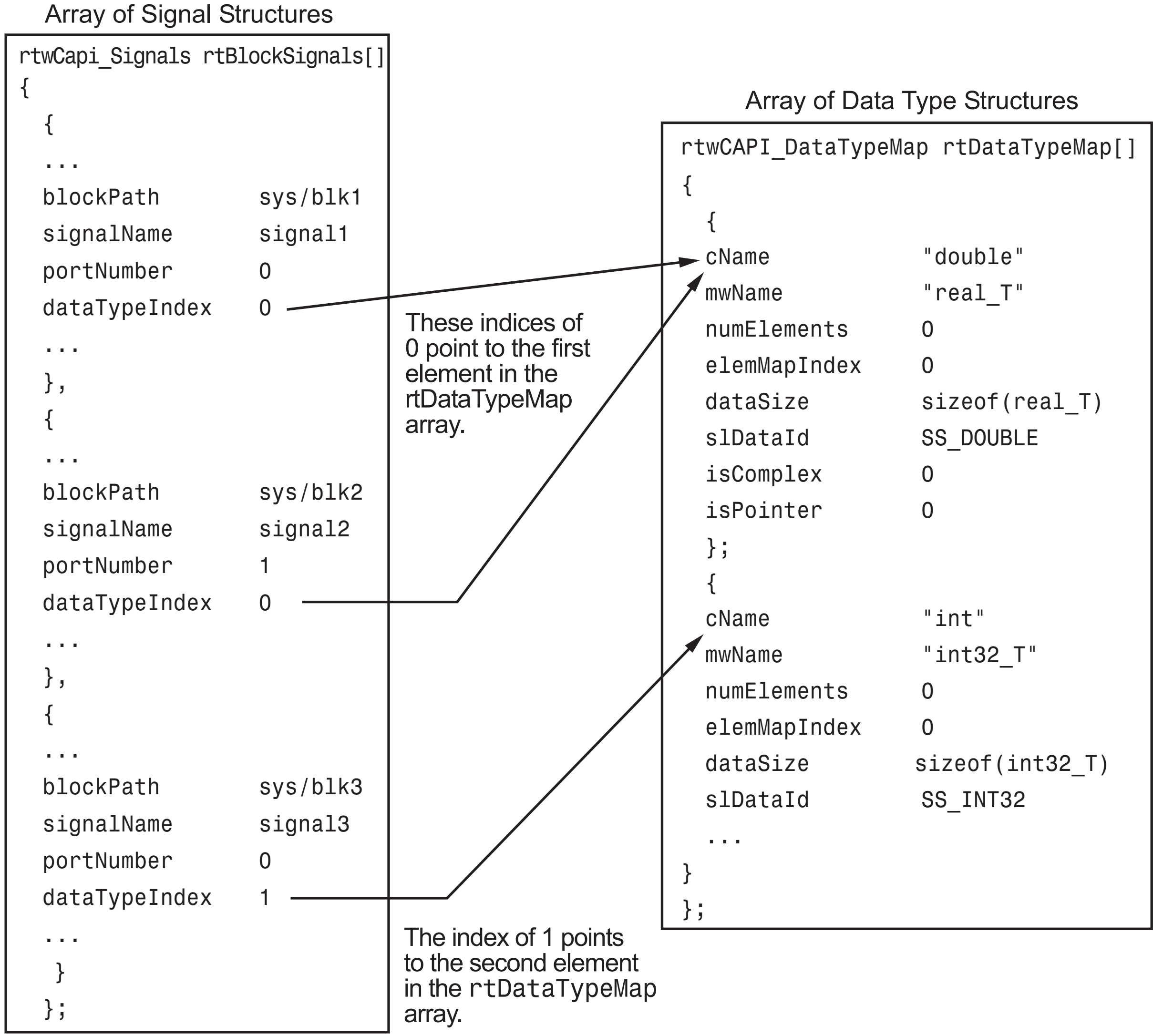
- Port number (for signals and root-level inputs/outputs only)
- Address
- Data type information: native data type, data size, complexity, and other attributes
- Dimensions information: number of rows, number of columns, and data orientation (scalar, vector, matrix, or *n*-dimensional)
- Fixed-point 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` (where *`matlabroot`* represents the root of your MATLAB installation folder). 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`):

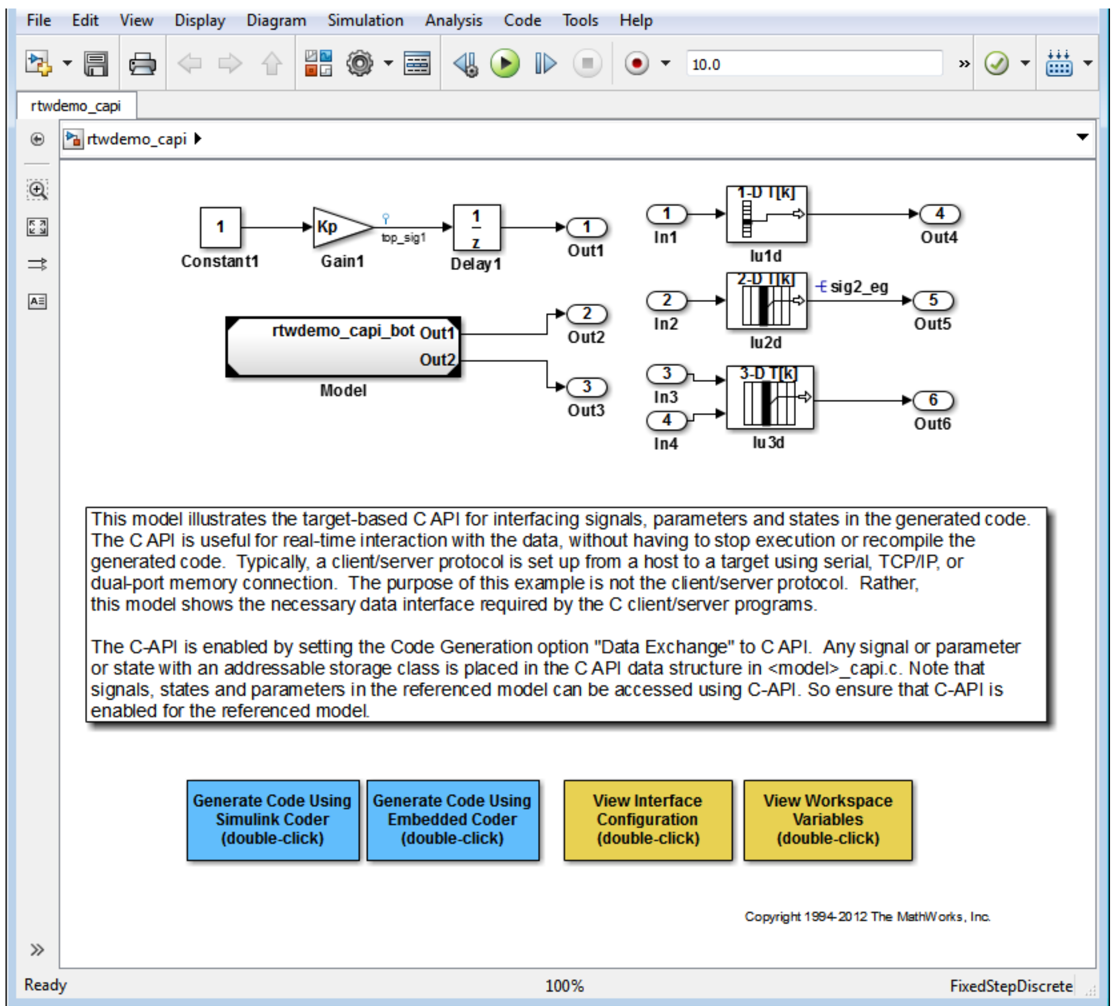
- **`rtBlockSignals`** is an array of structures that contains information about global block output signals in the model. Each element in the array is of type `struct rtwCAPI_Signals`. The members of this structure provide the signal name, block path, block port number, address, and indices to the data 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 rtwCAPI_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 rtwCAPI_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 rtwCAPI_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 rtwCAPI_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.

- **`rtModelParameters`** 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 `rtwCAPI_ModelParameters`. The members of this structure provide the variable name, address, and indices to data type, dimension, and fixed-point structure arrays.
- **`rtDataAddrMap`** is an array of base addresses of signals, states, root-level inputs/outputs, and parameters that appear in the `rtBlockSignals`, `rtBlockParameters`, `rtBlockStates`, and `rtModelParameters` arrays. Each element of the `rtDataAddrMap` array is a pointer to `void (void*)`.
- **`rtDataTypeMap`** 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 rtwCAPI_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.
- **`rtDimensionMap`** 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 rtwCAPI_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.
- **`rtFixPtMap`** 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 rtwCAPI_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 `rtFixPtMap` array.
- **`rtSampleTimeMap`** 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 rtwCAPI_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

The next three sections, “C API Signals” on page 17-136, “C API States” on page 17-139, “C API Root-Level Inputs and Outputs” on page 17-140, and “C API Parameters” on page 17-141, discuss generated C API structures using the example model rtwdemo_capi as an example. 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. The model appears as shown in the next figure.



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 make sure that the option **Generate C API for: root-level I/O** is selected.

Note: The setting of **Generate C API for: root-level I/O** must match between the top model and the referenced model.

3 Generate code for the model by double-clicking **Generate Code Using Simulink Coder**.

Note: The C API code examples in the next four sections 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" 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 17-136, 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 17-136.)

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, fxpIndex, sTimeIndex
   */
  { 3, 0, "rtwdemo_capi/In1",
    "", 1, 0, 0, 0, 0 },

  { 4, 0, "rtwdemo_capi/In2",
    "", 2, 0, 0, 0, 0 },

  { 5, 0, "rtwdemo_capi/In3",
    "", 3, 0, 0, 0, 0 },

  { 6, 0, "rtwdemo_capi/In4",
    "", 4, 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 17-136.

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. Each element in an `rtBlockParameters` or `rtModelParameters` array (except the last element) corresponds to a tunable parameter in the model.

The setting of the **Inline parameters** option on the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box determines how information is generated into the `rtBlockParameters` and `rtModelParameters` arrays in *`model`*`_capi.c` (or `.cpp`), as follows:

- If you clear **Inline parameters**:
  - The `rtBlockParameters` array contains an entry for every modifiable parameter of every block in the model.
  - The `rtModelParameters` array contains only Stateflow data of machine scope. The Simulink Coder software assigns its elements only `NULL` or zero values in the absence of such data.
- If you select **Inline parameters**:
  - The `rtBlockParameters` array is empty. The Simulink Coder software assigns its elements only `NULL` or zero values.
  - The `rtModelParameters` array contains entries for workspace variables that are referenced as tunable Simulink block parameters or Stateflow data of machine scope.

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 the **Inline parameters** option is selected by default for the `rtwdemo_capi` example model. If you clear this check box, the block parameters are generated in the `rtwCAPI_BlockParameters` 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 */
  { 3, "Ki", 0, 0, 0 },

  { 4, "Kp", 0, 0, 0 },

  { 5, "p1", 0, 2, 0 },

  { 6, "p2", 0, 3, 0 },

  { 7, "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 17-136, as follows:

- The address of the fourth parameter is given by `addrMapIndex`, which, in this example, is `6`. This is an index into the `rtDataAddrMap` array, found later in

rtwdemo_capi_capi.c. Because the index is zero based, 6 corresponds to the seventh element in rtDataAddrMap, which is &rtwP_p2[0].

- 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.

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;
```

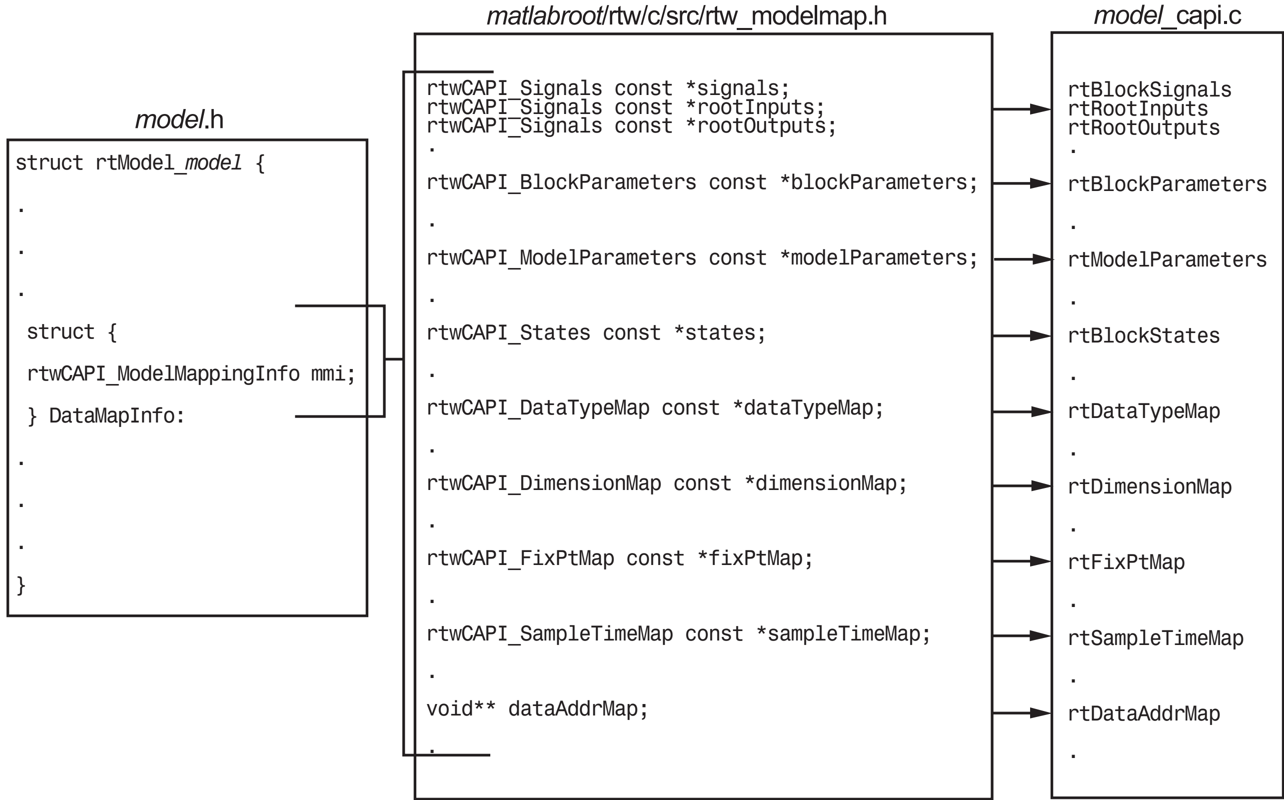
This member defines mmi (for model mapping information) of type struct rtwCAPI_ModelMappingInfo. The structure is located in *matlabroot*/rtw/c/src/rtw_modelmap.h (where *matlabroot* represents the root of your MATLAB installation folder). The mmi substructure defines the interface between the model 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`.

Use the C API in an Application

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 (where *matlabroot* represents the root of your MATLAB installation folder), provide macros for accessing the structures in these arrays and their members.

This section provides examples to help you get started writing application code to interact with model signals, states, root-level inputs/outputs, and parameters.

- "Use C API to Access Model Signals and States" on page 17-146
- "Use C API to Access Model Parameters" on page 17-153

Use C API to Access Model Signals and States

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, where *matlabroot* represents the root of your MATLAB installation folder.

  - `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. Open the Configuration Parameters dialog box.
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 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`.

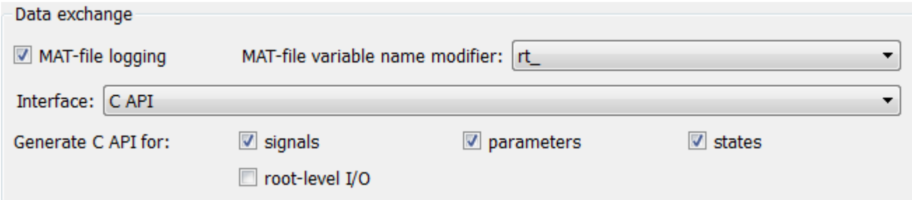
   Note: Selecting a system target file other than `grt.tlc` or disabling some C API options (signals, parameters, or states) in the top model requires corresponding changes in the referenced model. Because the example models have read-only access, you must save the updated referenced model with a different name and modify the top model to reference the renamed model.

   ---

4. Go to the **Code Generation** > **Interface** pane.

   a. In the **Data exchange** subpane, for the **Interface** parameter, verify that `C API` is selected.

   b. Verify that the options **Generate C API for: signals**, **Generate C API for: states**, and **MAT-file logging** are selected. This example also leaves **Generate C API for: parameters** selected.



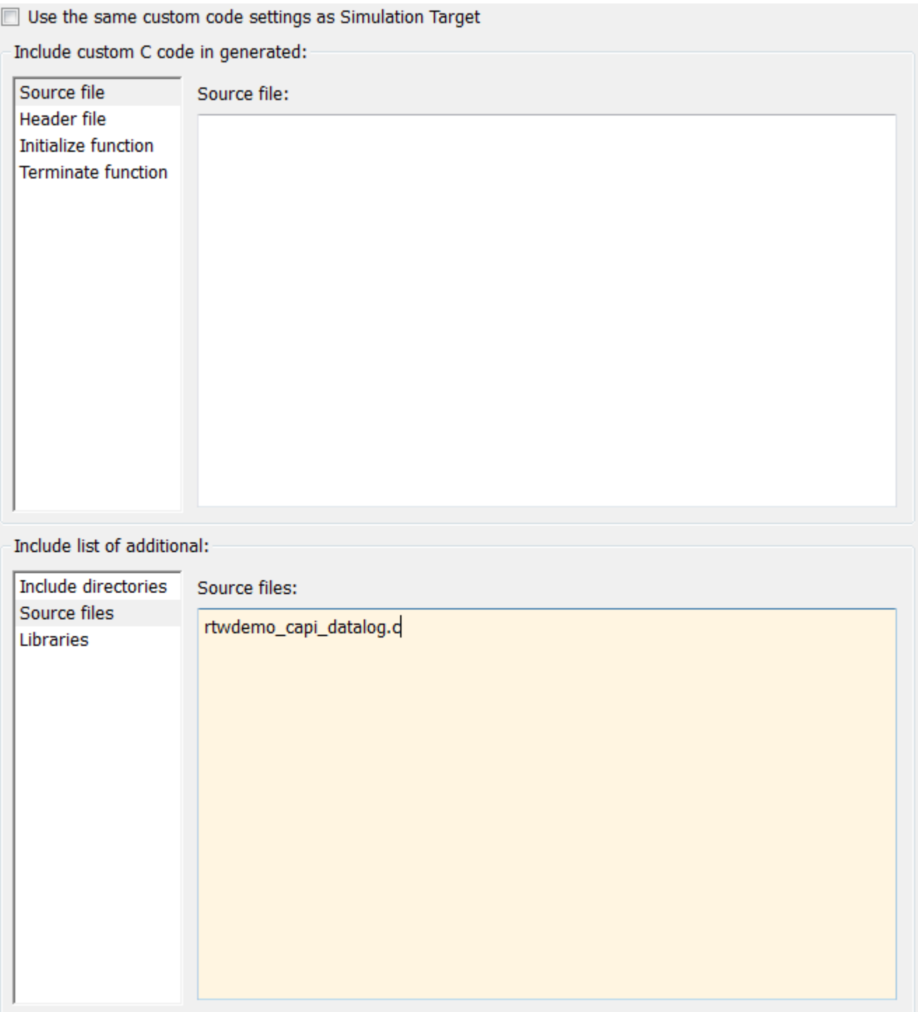
c If you are using the `ert.tlc` target, verify that the option **Support: complex numbers** is selected.

d If you modified an option setting in this step, click **Apply**. Also, update the option settings in the referenced model 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 **Include directories**. The **Include directories** input field is displayed.

6 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.)

7 In the **Include list of additional** subpane, click **Source files**, and type `rtwdemo_capi_datalog.c`, as shown below.



8 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"
```

9 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 string `rtwdemo_capi_M` in the application code to reflect the new model name.

10 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**.

11 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.

12 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 string `rtwdemo_capi_M` in the application code to reflect the new model name.

Click **OK**.

13 On the **Code Generation** pane, verify that the **Build** button is visible. If it is not visible, clear the option **Generate code only** and click **Apply**.

Click **Build** 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.

14 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 2 state(s). In this demo, only scalar named
 signals/states are logged **
** Finished state/signal logging. Created rtwdemo_capi_ModelLog.txt **
```

15 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: 2
Number of points (time steps) logged: 51

Time           bot_state (Referenced Model)           top_state
0              0                                      0
0.2            70                                     4
0.4            35                                     4
0.6            52.5                                   4
0.8            43.75                                  4
1              48.13                                  4
1.2            45.94                                  4
1.4            47.03                                  4
```

```
1.6         46.48                             4
1.8         46.76                             4
2           46.62                             4
...
```

Use C API to Access Model Parameters

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` (where *`matlabroot`* represents the root of your MATLAB installation folder). 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 Open the Configuration Parameters dialog box and go to the **Optimization** > **Signals and Parameters** pane.
3 Verify that the **Inline parameters** option is selected.
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`.

   Note: Selecting a system target file other than `grt.tlc` or disabling some C API options (signals, parameters, or states) in the top model requires corresponding changes in the referenced model. Because the example models have read-only access, you must save the updated referenced model with a different name and modify the top model to reference the renamed model.

   ---
5 Go to the **Code Generation** > **Interface** pane.
   a In the **Data exchange** subpane, for the **Interface** parameter, verify that `C API` is selected.

b. Verify that the options **Generate C API for: parameters** and **MAT-file logging** are selected.

c. If you are using the `ert.tlc` target, verify that the option **Support: complex numbers** is selected.

d. If you modified an option setting in this step, click **Apply**. Also, update the option settings in the referenced model 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 **Initialize function**. The **Initialize function** input field is displayed.

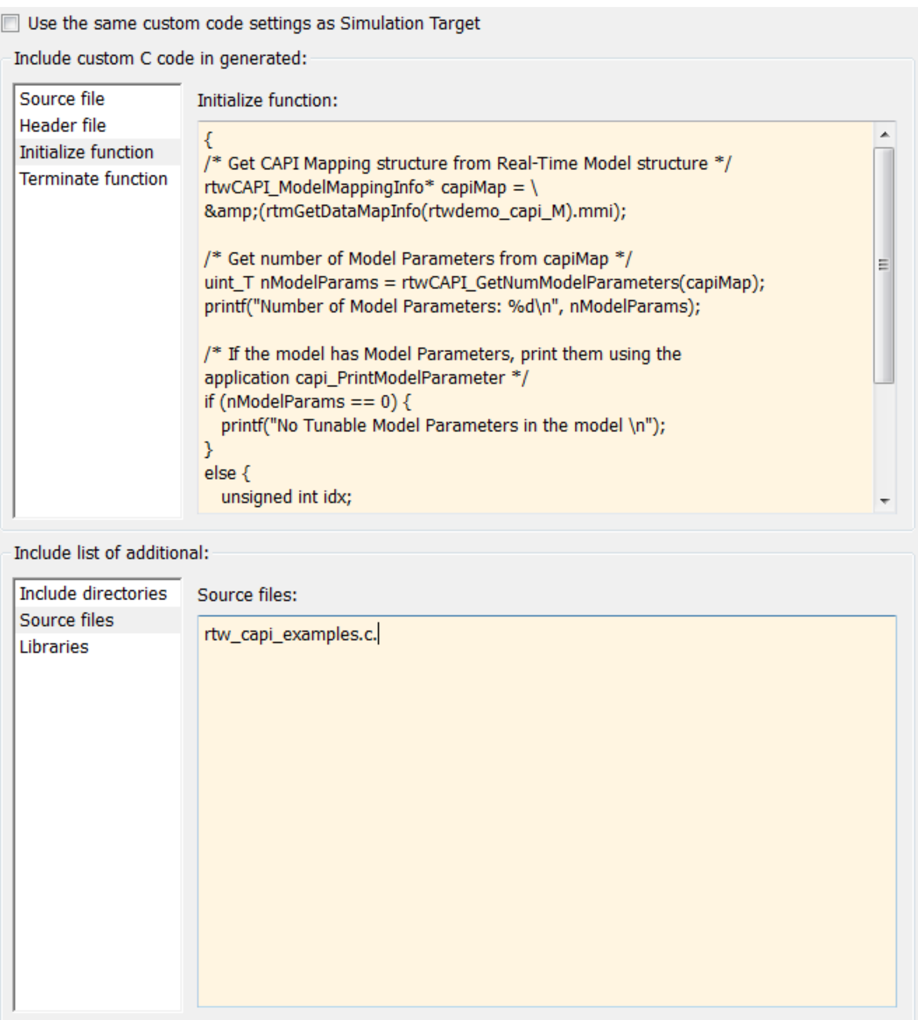
7. In the **Initialize function** input field, type or copy and paste the example application code shown above step 1. This embeds the application code in the *`model`*`_initialize` function.

Note: If you renamed the top model `rtwdemo_capi`, update the string `rtwdemo_capi_M` in the application code to reflect the new model name.

---

8. 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.)

9. In the **Include list of additional** subpane, click **Source files**, and type `rtw_capi_examples.c`.



Click **Apply**.

10 On the **Code Generation** pane, verify that the **Build** button is visible. If it is not visible, clear the option **Generate code only** and click **Apply**.

Click **Build** 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.

11 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
```

C API Limitations

The C API feature has the following limitations.

- The following code formats are not supported:
  - S-function
  - 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
  - `BitPackBoolean` objects, grouped custom storage classes, and objects defined by using macros
- 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.

ASAP2 Data Measurement and Calibration

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 ASAP2 standard, see the ASAM Web site at http://www.asam.net.

- "About ASAP2 Data Measurement and Calibration" on page 17-158
- "Targets Supporting ASAP2" on page 17-159
- "Define ASAP2 Information" on page 17-160
- "Generate an ASAP2 File" on page 17-165
- "Structure of the ASAP2 File" on page 17-169
- "Generate ASAP2 and C API Data Interfaces" on page 17-170

About ASAP2 Data Measurement and Calibration

The Simulink Coder product lets you export an ASAP2 file containing information about your model during the code generation process.

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 ”.

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.

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 17-165.

Define ASAP2 Information

- "Define ASAP2 Information for Parameters and Signals" on page 17-160
- "Memory Address Attribute" on page 17-161
- "Automatic ECU Address Replacement for ASAP2 Files (Embedded Coder)" on page 17-162
- "Define ASAP2 Information for Lookup Tables" on page 17-163

Define ASAP2 Information for Parameters and Signals

The ASAP2 file generation process requires information about your model's parameters and signals. 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`

Data Attribute	Defined In	Property Name
Maximum allowable value	Data object	`Max`
Units	Data object	`DocUnits`
Memory address (optional)	Data object	`MemoryAddress_ASAP2` (optional; see "Memory Address Attribute" on page 17-161.)

Memory Address Attribute

If the memory address attribute is unknown before code generation, the code generator inserts an `ECU Address` placeholder string 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 placeholder strings 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 17-162.

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)
LONGIG_ASAP2	Description
PhysicalMin_ASAP2	Min
PhysicalMax_ASAP2	Max
Units_ASAP2	DocUnits

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 an `ECU Address` placeholder string 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 strings 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`: • `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 17-165
- "Use GRT or ERT Target" on page 17-165
- "Use the ASAM-ASAP2 Data Definition Target" on page 17-167
- "Generate ASAP2 Files for Referenced Models" on page 17-168
- "Merge ASAP2 Files for Top and Referenced Models" on page 17-168

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 a model's data definition in ASAP2 format 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. Define the desired parameters and signals in the model to be `Simulink.Parameter` and `Simulink.Signal` objects in the MATLAB workspace. 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 "Data Object Wizard" in the Simulink documentation.

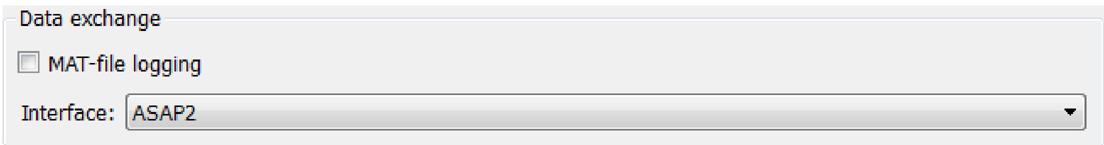
3. For each data object, configure the **Storage class** property to a setting other than `Auto` or `SimulinkGlobal`. This declares the data object as global in the generated code. For example, a storage class setting of `ExportedGlobal` configures the data object as unstructured global in the generated code.

Note: If you set the storage class to `Auto` or `SimulinkGlobal`, or if you set the storage class to `Custom` and custom storage class settings cause the Simulink Coder code generator to generate a macro or non-addressable variable, the data object is not represented in the ASAP2 file.

---

4. Configure the remaining properties as desired for each data object.
5. On the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box, select the **Inline parameters** check box.

   You should *not* configure the parameters associated with your data objects as Simulink global (tunable) parameters in the Model Parameter Configuration dialog box. If a parameter that resolves to a Simulink data object is configured using the Model Parameter Configuration dialog box, the dialog box configuration is ignored. You can, however, use the Model Parameter Configuration dialog box to configure other parameters in your model.
6. On the **Code Generation** pane, click **Browse** to open the System Target File Browser. In the browser, select `Generic Real-Time Target` or another embedded real-time target and click **OK**.
7. In the **Interface** field on the **Interface** pane, select `ASAP2`.

8. Select the **Generate code only** check box on the **Code Generation** pane.
9. Click **Apply**.
10. Click **Generate Code**.

    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 filename is controlled by the ASAP2 setup file. For details see "Customize an ASAP2 File" on page 25-2.

Use the ASAM-ASAP2 Data Definition Target

The procedure for generating a model's data definition in ASAP2 format 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. Define the desired parameters and signals in the model to be `Simulink.Parameter` and `Simulink.Signal` objects in the MATLAB workspace. 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 "Data Object Wizard" in the Simulink documentation.
3. For each data object, configure the **Storage class** property to a setting other than `Auto` or `SimulinkGlobal`. This causes the data object to be declared 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: If you set the storage class to `Auto` or `SimulinkGlobal`, or if you set the storage class to `Custom` and custom storage class settings cause the Simulink Coder code generator to generate a macro or non-addressable variable, the data object is not represented in the ASAP2 file.

   ---

4. Configure the remaining properties as desired for each data object.
5. On the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box, select the **Inline parameters** check box.

   You should *not* configure the parameters associated with your data objects as Simulink global (tunable) parameters in the Model Parameter Configuration dialog box. If a parameter that resolves to a Simulink data object is configured using the Model Parameter Configuration dialog box, the dialog box configuration is ignored. You can, however, use the Model Parameter Configuration dialog box to configure other parameters in your model.
6. On the **Code Generation** pane, click **Browse** to open the System Target File Browser. In the browser, select `ASAM-ASAP2 Data Definition Target` and click **OK**.

7. Select the **Generate code only** check box on the **Code Generation** pane.
8. Click **Apply**.
9. Click **Generate Code**.

   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 filename is controlled by the ASAP2 setup file. For details see "Customize an ASAP2 File" on page 25-2.

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 17-165 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 string.

Consider the following example.

```
[status,info]=rtw.asap2MergeMdlRefs('myTopMdl','merged.a2l')
```

This command merges the ASAP2 files generated for the top model `myTopMdl` and its referenced models in the file `merged.a2l`.

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 asap2main.tlc	TLC File Containing Function Definition
File header	%<ASAP2UserFcnWriteFileHead()>	asap2userlib.tlc
/**begin PROJECT** ""	/**begin PROJECT** "%<ASAP2ProjectName>"	asap2setup.tlc
/**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**	

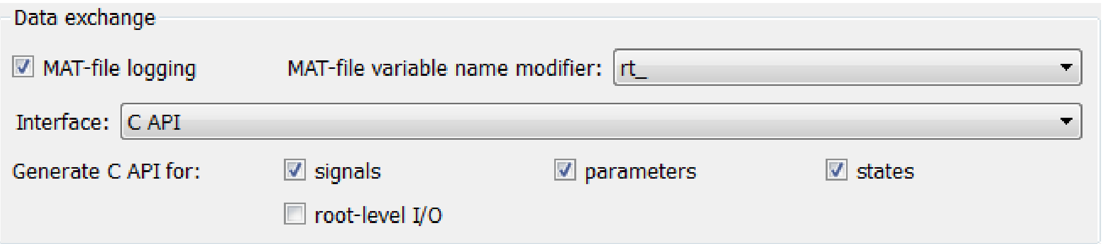
File Section	Contents of asap2main.tlc	TLC File Containing Function Definition
File footer/tail	`%<ASAP2UserFcnWriteFileTail()>`	`asap2userlib.tlc`

Generate ASAP2 and C API Data Interfaces

The ASAP2 and C API data interfaces for model code are not mutually exclusive. You can generate code for your model with both the ASAP2 and C API interfaces enabled.

To configure your model so that the build process generates files for both the ASAP2 and C API interfaces, you must configure at least one of the data interfaces at the MATLAB command line. The following example configures the C API in the Configuration Parameters dialog box and configures the ASAP2 at the command line.

1 On the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box, select `C API` for the **Interface** parameter. Inspect the settings for the other C API related controls and make your adjustments. Click **Apply**.



2 In the MATLAB Command Window, with the model open, specify the following command:

```
>> set_param(gcs,'GenerateASAP2','on')
```

Note: If you select both the ASAP2 and C API data interfaces, the Configuration Parameters dialog box (in subsequent opens) displays only the C API options. You can display the current ASAP2 option setting using the `get_param` command.

---

3 Click **Generate Code** or **Build**. The build process generates the following ASAP2 and C API files:

- *`model`*`.a2l` — ASAP2 description file
- *`model`*`_capi.c` — C API source file

- `*model*_capi.h` — C API header file

For more information about using the C API interface, see "Data Interchange Using the C API" on page 17-125.

Direct Memory Access to Generated Code

The Simulink Coder product provides a 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

- "Optimization Parameters" on page 18-2
- "Advice About Optimizing Models for Code Generation" on page 18-4
- "Control Compiler Optimizations" on page 18-5
- "Optimization Tools and Techniques" on page 18-6
- "Control Memory Allocation for Time Counters" on page 18-8
- "Optimization Parameter Dependencies" on page 18-9
- "Execution Profiling for Generated Code" on page 18-11

Optimization Parameters

Many parameters, on the **Optimization** and **Optimization** > **Signals and Parameters** panes of the Configuration Parameters dialog box, depending on their setting, modify the generated code. For more information on these parameters, see

- "Optimization Parameter Dependencies" on page 18-9
- "Optimization Pane: General"
- "Optimization Pane: Signals and Parameters".

Some basic optimization suggestions are given below.

- 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" on page 22-4.
- Selecting the **Inline parameters** check box reduces global RAM usage, because parameters are not declared in the global parameters structure. You can override the inlining of individual parameters by using the Model Parameter Configuration dialog box. You tune parameters used in referenced models differently, by declaring them as Model block parameter arguments, rather than using the Model Parameter Configuration dialog box. For more information, see "Inline Parameters" on page 21-2 and "Using Model Arguments" in the Simulink documentation.
- 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**. For more information, see "Declare Signals as Local Function Data" on page 20-13.
- 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**. For more information, see "Reuse Memory Allocated for Signals" on page 22-5.
- 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 select **Inline parameters** to enable **Inline invariant signals**. For more information, see "Inline Invariant Signals" on page 20-14.
- 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" on page 20-8.

- Selecting the **Minimize data copies between local and global variables** check box reuses existing global variables to store temporary results. You must select **Signal storage reuse** to enable **Minimize data copies between local and global variables**. For more information, see "Reuse Memory Allocated for Signals" on page 22-5.
- 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 21-4.
- 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 22-6.
- 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 "Optimize Code Generated for Vector Assignments" on page 21-6.

To view examples that illustrate optimization settings and techniques, at the Command Window, type

```
rtwdemos
```

and navigate to the **Optimizations** section.

Advice About Optimizing Models for Code Generation

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 targets (targets based on the Embedded Coder software).

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.

Control Compiler Optimizations

To control compiler optimizations for your Simulink Coder makefile build at the Simulink 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 Simulink Coder 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`).

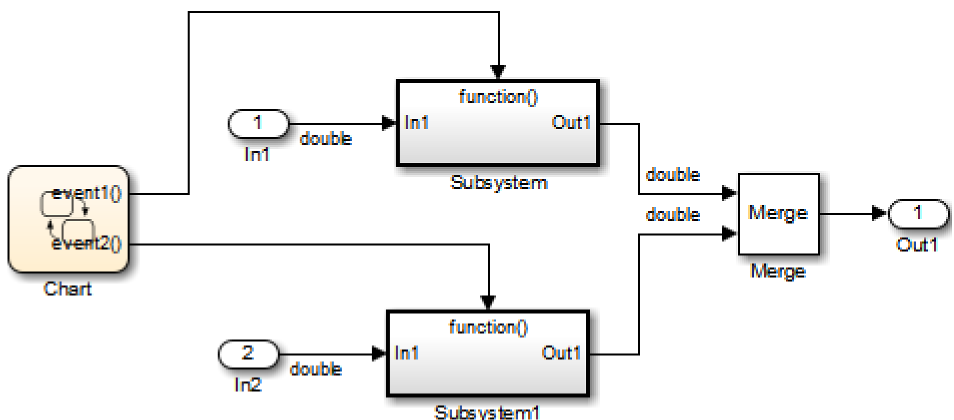
Note: If you specify compiler options for your Simulink Coder makefile build using `OPT_OPTS`, `MEX_OPTS` (except `MEX_OPTS="-v"`), or `MEX_OPT_FILE`, the value of **Compiler optimization level** is ignored and a warning is issued about the ignored parameter.

For more information about the **Compiler optimization level** parameter and its values, see “Compiler optimization level” and “Custom compiler optimization flags”.

Optimization Tools and Techniques

In addition to analyzing models with Model Advisor (see “Advice About Optimizing Models for Code Generation” on page 18-4), you can use a variety of other tools and techniques. Here are some particularly useful ones:

- 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 “Generated S-Function Block” on page 14-31 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 “Data Types”. For a block-by-block summary, click showblockdatatypetable or type the command in the Command Window.
- 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.

If you are licensed to use the Embedded Coder product, for more information, see "Performance" in the Embedded Coder documentation.

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 “Limitations on the Use of Absolute Time” on page 1-78.

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 Simulink Coder length limitation of 32 bits because the value of a time counter does not provide the value of a signal, state, or parameter.

For information about the allocation and operation of absolute and elapsed time counters, see “Timers” on page 1-69. For information about asynchronous timing, see “Use Timers in Asynchronous Tasks” on page 1-51. For information about the **Application lifespan (days)** parameter, see “Application lifespan (days)” in the Simulink documentation.

Optimization Parameter Dependencies

Several parameters available on the **Optimization** panes 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
Optimize initialization code for model reference	ERT targets only
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

Optimization > Signals and Parameters pane	Dependency Details
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**
Inline invariant signals	Enabled by **Inline parameters**
Minimize data copies between local and global variables	Enabled by **Signal storage reuse**
Simplify array indexing	ERT targets only
Memcpy threshold (bytes)	Enabled by **Use memcpy for vector assignment**
Pack Boolean data into bitfields	ERT targets only
Bitfield declarator type specifier	• Enabled by **Pack Boolean data into bitfields** • ERT targets only

Optimization > Signals and Parameters pane	Dependency Details
Pass reusable subsystem output as	ERT targets only
Parameter structure	• Enabled by **Inline parameters** • ERT targets only

For more information on these parameters, see “Optimization Pane: General” and “Optimization Pane: Signals and Parameters”.

Execution Profiling for Generated Code

Use code execution profiling to determine:

- Whether the generated code meets real-time requirements of your target hardware.
- Code sections that require performance improvements.

The following tasks represent a general workflow that uses code execution profiling:

1. With the Simulink model, design and optimize your algorithm.
2. Configure the model for code execution profiling, and generate code.
3. Execute generated code on target.
4. Analyze performance through code execution profiling plots and reports. For example, check that the algorithm code satisfies real-time requirements:
   - If the algorithm code easily meets the requirements, consider enhancing your algorithm to exploit available processing power.
   - If the code cannot be executed in real time, look for ways to reduce execution time.

     Identify the tasks that require the most time. For these tasks, investigate whether trade-offs between functionality and speed are possible.

     If your target is a multicore processor, distribute the execution of algorithm code across available cores.
5. If required, refine the model and return to step 2.

To find information about code execution profiling with Simulink products, use the following table.

Target	Execution Feature	Type of Profiling	Relevant Products	See
Host computer	Model configured for concurrent execution	Execution time	Simulink Coder	• “Build and Download to a Multicore Target” • “Concurrent Execution Example Models”
Host computer	Software-in-the-loop (SIL)	Execution time	Embedded Coder	• “Code Execution Profiling for SIL and PIL”

Target	Execution Feature	Type of Profiling	Relevant Products	See
				• “Configure Code Execution Profiling for SIL and PIL” • “Execution Profiling for Atomic Subsystems and Model Reference Hierarchies” • “View and Compare Code Execution Times” • “Analyze Code Execution Data”
Embedded hardware or instruction set simulator	Processor-in-the-loop (PIL)	Execution time	Embedded Coder	• “Code Execution Profiling for SIL and PIL” • “Configure Code Execution Profiling for SIL and PIL” • “Execution Profiling for Atomic Subsystems and Model Reference Hierarchies” • “View and Compare Code Execution Times” • “Analyze Code Execution Data”
Target support packages	Standalone execution, PIL	Execution time	Embedded Coder	• “Code Execution Profiling for IDE and Toolchain Targets” • “Perform Execution Time Profiling for IDE and Toolchain Targets”
Target support packages	Standalone execution	Stack	Embedded Coder	• “Code Execution Profiling for IDE and Toolchain Targets” • “Perform Stack Profiling with IDE and Toolchain Targets”

Target	Execution Feature	Type of Profiling	Relevant Products	See
Simulink Real-Time	Hardware-in-the-loop (HIL)	Execution time	Simulink Coder, Simulink Real-Time	• “Execution Profiling for Real-Time Applications” • “Configure Real-Time Application for Profiling” • “Generate Real-Time Application Execution Profile”
Simulink Real-Time	Model configured for concurrent execution, HIL	Execution time	Simulink Coder, Simulink Real-Time	• “Execution Profiling for Real-Time Applications” • “Concurrent Execution with Simulink® Real-Time™”

Defensive Programming

- "Optimize Code for Floating-Point to Integer Conversions" on page 19-2
- "Disable Nonfinite Checks or Inlining for Math Functions" on page 19-4

Optimize Code for Floating-Point to Integer Conversions

In this section...
"Remove Code That Wraps Out-of-Range Values" on page 19-2
"Remove Code That Maps NaN to Integer Zero" on page 19-2

Remove Code That Wraps Out-of-Range Values

Selecting the **Remove code from floating-point to integer conversions that wraps out-of-range values** check box in the **Integer and fixed-point** section of the **Optimization** pane causes the code generator to remove code that produces the same results as simulation when out-of-range conversions occur. This action reduces the size and increases the speed of generated code at the cost of potentially producing results that do not match simulation in the case of out-of-range values. For more information, see "Optimization Pane: General".

The code generated for a conversion when you select the check box follows:

```
cg_in_0_20_0[i1] = (int32_T)((rtb_Switch[i1] + 9.0) / 0.09375);
```

The code generated for a conversion when you clear the check box follows:

```
_fixptlowering0 = (rtb_Switch[i1] + 9.0) / 0.09375;
   _fixptlowering1 = fmod(_fixptlowering0 >= 0.0 ? floor(_fixptlowering0) :
      ceil(_fixptlowering0), 4.2949672960000000E+009);
    if(_fixptlowering1 < -2.1474836480000000E+009) {
      _fixptlowering1 += 4.2949672960000000E+009;
    } else if(_fixptlowering1 >= 2.1474836480000000E+009) {
      _fixptlowering1 -= 4.2949672960000000E+009;
    }
    cg_in_0_20_0[i1] = (int32_T)_fixptlowering1;
```

The code generator applies the `fmod` function to handle out-of-range conversion results.

Remove Code That Maps NaN to Integer Zero

Selecting the **Remove code from floating-point to integer conversions with saturation that maps NaN to zero** check box in the **Integer and fixed-point** section of the **Optimization** pane causes the code generator to remove code that produces the same results as simulation when mapping from `NaN` to integer zero occurs. This action reduces the size and increases the speed of generated code at the cost of producing

results that do not match simulation in the case of NaN values. For more information, see "Optimization Pane: General".

The code generated for a conversion when you select the check box follows:

```
if (tmp < 2.147483648E+09) {
  if (tmp >= -2.147483648E+09) {
    tmp_0 = (int32_T)tmp;
  } else {
    tmp_0 = MIN_int32_T;
  }
} else {
  tmp_0 = MAX_int32_T;
}
```

The code generated for a conversion when you clear the check box follows:

```
if (tmp < 2.147483648E+09) {
  if (tmp >= -2.147483648E+09) {
    tmp_0 = (int32_T)tmp;
  } else {
    tmp_0 = MIN_int32_T;
  }
} else if (tmp >= 2.147483648E+09) {
  tmp_0 = MAX_int32_T;
} else {
  tmp_0 = 0;
}
```

Disable Nonfinite Checks or Inlining for Math Functions

When Simulink Coder software generates 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.TflTable;

% 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 (`>> RTW.viewTfl(crl_table_customization)`). For more information about viewing code replacement tables, see "Choose a Code Replacement Library".
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.

Data Copy Reduction

- “Optimize Buffers in the Generated Code” on page 20-2
- “Minimize Computations and Storage for Intermediate Results” on page 20-8
- “Declare Signals as Local Function Data” on page 20-13
- “Inline Invariant Signals” on page 20-14

Optimize Buffers in the Generated Code

In this section...

"Configure Buffer Optimizations" on page 20-2
"Example Model" on page 20-2
"Generate Code Without Buffer Optimization" on page 20-3
"Enable Buffer Optimization" on page 20-5

Configure Buffer Optimizations

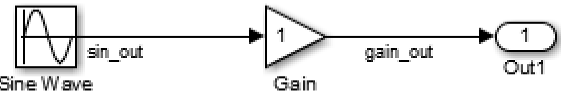
This example shows how to configure optimizations, using the parameters:

- **Signal storage reuse**
- **Enable local block outputs**
- **Reuse local block outputs**
- **Minimize data copies between local and global variables**

Buffer optimizations reduce RAM and ROM consumption and improve execution speed.

Note You can view the code generated from this example using the MATLAB editor or by generating an HTML code generation report. For more information, see "Generate a Code Generation Report".

This model is the example model.



Example Model

Create the model from Simulink library blocks, and select parameters as follows:

1. Create a temporary folder, `example_codegen`, and make it your working folder:

   ```
   !mkdir example_codegen
   cd example_codegen
   ```

2. In the Command Window, type `simulink`. Create a new model and save it as `example`.
3. Add Sine Wave, Gain, and Out1 blocks to your model. Connect them as shown in the preceding diagram. Label the signals as shown. Save.
4. From the **Simulation** menu, select **Model Configuration Parameters** to open the Configuration Parameters dialog box.
5. In the **Solver options** section:

   a. In the **Type** field, select `Fixed-step` .

   b. In the **Solver** field, select `discrete (no continuous states)`.

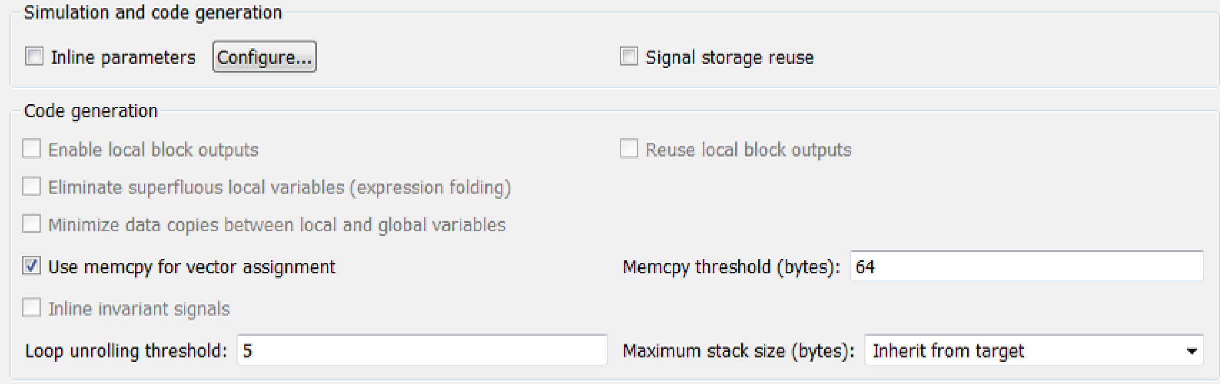
   c. In the **Fixed-step size** field, replace `auto` with `0.1`.

   Click **Apply**.
6. In the left pane, click **Data Import/Export** Clear the check boxes in the right pane. Click **Apply**.
7. Select **Code Generation**. In the **System target file** list in the right pane, specify the default generic real-time (GRT) target `grt.tlc`.
8. Select **Generate code only** at the bottom of the left pane. This option causes the software to generate code and a makefile, and stop. The label on the **Build** button changes to **Generate Code**.
9. Click **Apply** and save the model.

Generate Code Without Buffer Optimization

Disable the buffer optimizations to generate the unoptimized code.

1. In the Configuration Parameters dialog box, select **Optimization** > **Signals and Parameters**. Clear the **Signal storage reuse** check box, and select options as shown.



2. Click **Apply**.
3. Select **Code Generation** > **Report**.
4. Select the **Create code generation report** and **Open report automatically** check boxes. After the code generation process completes, the code generation report displays.
5. Click **Apply**.
6. On the **Code Generation** pane, click **Generate Code**.

   Because you selected the **Generate code only** option, the build process does not invoke your `make` utility. The code generation process ends with this message in the Diagnostic Viewer:

   ```
   Build process completed successfully
   ```
7. The generated code is in the build folder, `example_grt_rtw`. The file `example_grt_rtw/example.c` contains the output computation for the model. You can view this file in the code generation report by clicking the `example.c` link in the left pane.
8. In `example.c`, find the function `example_step` near the top of the file.

   The generated C code consists of procedures that implement the algorithms defined by your model. The execution engine calls the procedures in order.

   In code generated for `example`, the generated `example_step` function implements the actual algorithm for multiplying a sine wave by a gain. The `example_step`

function computes the model block outputs. The run-time interface must call `example_step` at every time step.

With buffer optimizations turned off, `example_step` assigns unique buffers to each block output. These buffers, example_B.sin_out and example_B.gain_out, are members of a global block I/O data structure, example_B, and declared as follows:

```
/* Block signals (auto storage) */
B_example_T example_B;
```

The structures `example_P`, and `example_Y` are defined in `example.h` as global structures:

```
/* Block parameters (auto storage) */
extern P_example_T example_P;

/* External outputs (root outports fed by signals with auto storage) */
extern ExtY_example_T example_Y;
T;
```

The output code accesses fields of these global structures. The results of the calculation are assigned to `example_B.sin_out` and then copied to `example_Y.Out1` as shown:

```
void example_step(void)
{
  /* Sin: '<Root>/Sine Wave' */
  example_B.sin_out = sin(example_P.SineWave_Freq * example_M->Timing.t[0] +
    example_P.SineWave_Phase) * example_P.SineWave_Amp + example_P.SineWave_Bias;

  /* Gain: '<Root>/Gain' */
  example_B.gain_out = example_P.Gain_Gain * example_B.sin_out;

  /* Outport: '<Root>/Out1' */
  example_Y.Out1 = example_B.gain_out;
```

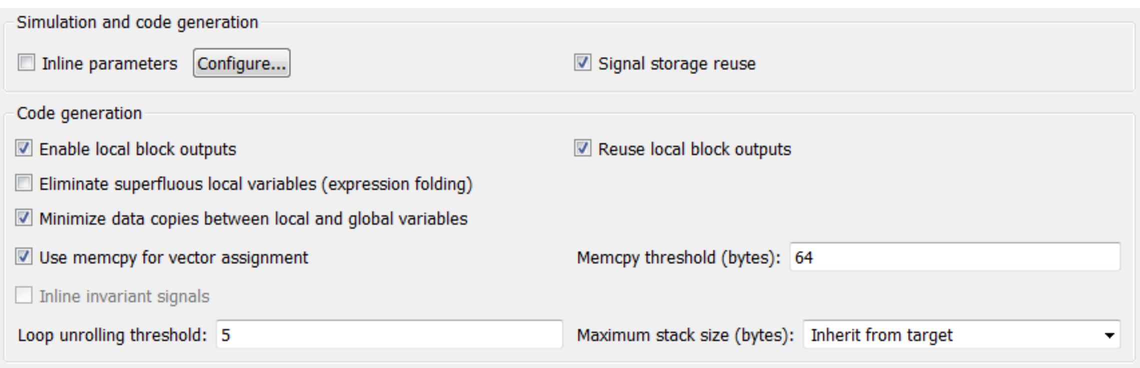
Enable Buffer Optimization

With buffer optimizations, the software generates code that reduces ROM and RAM consumption and improves execution speed. In this example, you turn buffer optimizations on and observe how they improve the code. Enable signal buffer optimizations and regenerate the code as follows:

1. In the Configuration Parameters dialog box, select **Optimization** > **Signals and Parameters**.
2. Select these check boxes:

- **Signal storage reuse**
- **Enable local block outputs**
- **Reuse local block outputs**
- **Minimize data copies between local and global variables**

3. Clear the **Eliminate superfluous local variables (expression folding)** check box.



4. Click **Apply**.
5. Select **Code Generation**. Click **Generate Code**. The previously generated code is overwritten.
6. View `example.c` and examine the `example_step` function.

With buffer optimizations enabled, the code in `example_step` reuses the `example_Y.Out1` global buffer. Now, `example_B` is not used.

```
void example_step(void)
{
  /* Sin: '<Root>/Sine Wave' */
  example_Y.Out1 = sin(example_P.SineWave_Freq * example_M->Timing.t[0] +
                       example_P.SineWave_Phase) * example_P.SineWave_Amp +
    example_P.SineWave_Bias;

  /* Gain: '<Root>/Gain' */
  example_Y.Out1 *= example_P.Gain_Gain;
```

This code reduces ROM and RAM consumption and improves execution speed. By eliminating a global variable and a data copy instruction, the data section is smaller and the code is smaller and faster. The efficiency improvement gained by selecting **Enable**

local block outputs, **Reuse block outputs**, and **Minimize data copies between local and global variables** is more significant in a large model with many signals.

With an Embedded Coder license, if you select an embedded target such as `ert.tlc`, the software replaces the **Minimize data copies between local and global variables** check box with the **Optimize global data access** list. When **Minimize data copies between local and global variables** is selected, **Optimize global data access** is set to `Use global to hold temporary results`.

Minimize Computations and Storage for Intermediate Results

In this section...
"About Expression Folding" on page 20-8
"Expression Folding Example" on page 20-9
"Enable Expression Folding" on page 20-11

About Expression Folding

Expression folding is a code optimization technique that minimizes the computation of intermediate results at block outputs and the storage of such results in temporary buffers or variables. **Eliminate superfluous local variables (expression folding)** is used to enable expression folding. When expression folding is on, the Simulink Coder code generator collapses, or "folds," block computations into a single expression, instead of generating separate code statements and storage declarations for each block in the model.

Expression folding can dramatically improve 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.

By default, expression folding is on. The Simulink Coder code generation options are configured to use expression folding wherever possible. Most Simulink blocks support expression folding.

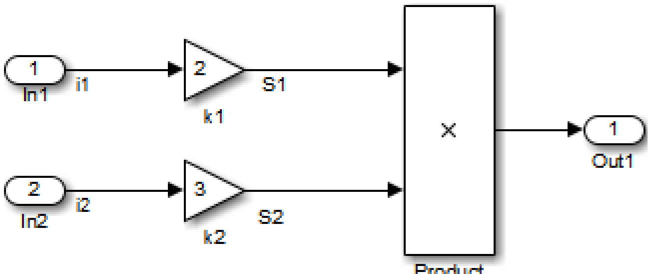
You can also take advantage of expression folding in your own inlined S-function blocks. See "Write S-Functions That Support Expression Folding" on page 16-76 for information on how to do this.

In the code generation examples that follow, the **Signal storage reuse** optimizations listed below are selected:

- **Enable local block outputs**
- **Reuse local block outputs**
- **Eliminate superfluous local variables (expression folding)**
- **Minimize data copies between local and global variables**

Expression Folding Example

As a simple example of how expression folding affects the code generated from a model, consider the following model.



With expression folding on, this model generates a single-line output computation, as shown in this *model*`_output` function.

```
static void exprfld_output(int_T tid)
{

  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/k1'
   *  Gain: '<Root>/k2'
   *  Inport: '<Root>/In1
   *  Inport: '<Root>/In2
   *  Product: '<Root>/Product'
   */
  exprfld_Y.Out1 = exprfld_U.i1 * exprfld_P.k1_Gain *
    (exprfld_U.i2 * exprfld_P.k2_Gain);
}
```

The generated comments indicate the block computations that were combined into a single expression. The comments also document the block parameters that appear in the expression.

With expression folding off, the same model computes temporary results for both Gain blocks before the final output, as shown in this output function:

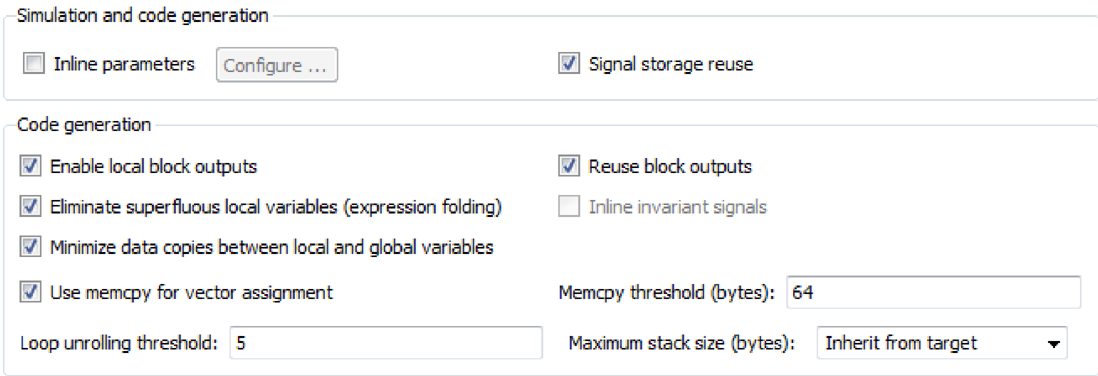
```
static void exprfld_output(int_T tid)
{
```

```
  real_T rtb_S2;

  /* Gain: '<Root>/k1' incorporates:
   * Inport: '<Root>/In1'
   */
  exprfld_Y.Out1 = exprfld_U.i1 * exprfld_P.k1_Gain

  /* Gain: '<Root>/k2' incorporates:
   * Inport: '<Root>/In2'
   */
  rtb_S2 = exprfld_U.i2 * exprfld_P.k2_Gain;

  /* Product: '<Root>/Product' */
  exprfld_Y.Out1 = exprfld_Y.Out1 * rtb_S2;
}
```

Turn on expression folding, a code optimization technique that minimizes the computation of intermediate results and the use of temporary buffers or variables.

Enable expression folding and regenerate the code as follows:

1. Change your current working folder to `example_codegen` if you have not already done so.
2. In Model Explorer, select **Optimization** in the center pane and click the **Signals and Parameters** tab on the right pane. The **Signals and Parameters** pane appears at the right.
3. Select the **Eliminate superfluous local variables (expression folding)** option.



4. Click **Apply**.

5 Select **Code Generation** in the center pane, and click **Generate Code** on the right.

   The Simulink Coder software generates code as before.

6 View `example.c` and examine the function `example_output`.

In the previous examples, the Gain block computation was computed in a separate code statement and the result was stored in a temporary location before the final output computation.

With **Eliminate superfluous local variables (expression folding)** selected, there is a subtle but significant difference in the generated code: the gain computation is incorporated (or folded) directly into the Outport computation, eliminating the temporary location and separate code statement. This computation is on the last line of the `example_output` function.

```
/* Model output function */
static void example_output(int_T tid)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain'
   *  Sin: '<Root>/Sine Wave'
   */
  example_Y.Out1 = (sin(example_P.SineWave_Freq * example_M->Timing.t[0] +
                        example_P.SineWave_Phase) * example_P.SineWave_Amp +
                    example_P.SineWave_Bias) * example_P.Gain_Gain;

  /* tid is required for a uniform function interface.
  * Argument tid is not used in the function. */
  UNUSED_PARAMETER(tid);
}
```

In many cases, expression folding can incorporate entire model computations into a single, highly optimized line of code. Expression folding is turned on by default. Using this option will improve the efficiency of generated code.

For an example of expression folding in the context of a more complex model, click `rtwdemo_slexprfold` , or type the following command at the MATLAB prompt.

```
rtwdemo_slexprfold
```

Enable Expression Folding

Expression folding operates only on expressions involving local variables. Expression folding is therefore available only when the **Signal storage reuse** code generation option is on.

For a new model, default code generation options are set to use expression folding. If you are configuring an existing model, enable expression folding as follows:

1. Open the Configuration Parameters dialog box and select the **Optimization** > **Signals and Parameters** pane.
2. Select the **Signal storage reuse** check box.
3. Select the **Enable local block outputs** check box.
4. Select the **Reuse local block outputs** check box.
5. Select the **Minimize data copies between local and global variables** check box.
6. Enable expression folding by selecting **Eliminate superfluous local variables (expression folding)**.

   Expression folding related options are now selected.
7. Click **Apply**.

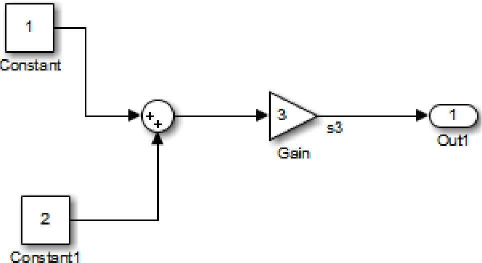
Declare Signals as Local Function Data

To declare block signals locally in functions instead of being declared globally (when possible), select the **Enable local block outputs** configuration parameter. This parameter is available only when you select **Signal storage reuse**.

For more information on the use of the **Enable local block outputs** option, see "Signals" on page 8-46.

Inline Invariant Signals

An invariant signal is a block output signal that does not change during Simulink simulation. For example, the signal S3 in this block diagram is an invariant signal.



For the previous model, if you select **Inline invariant signals** on the **Optimization** > **Signals and Parameters** pane, the Simulink Coder code generator inlines the invariant signal S3 in the generated code.

Note that an *invariant signal* is not the same as an *invariant constant*. In the preceding example, the two constants (1 and 2) and the gain value of 3 are invariant constants. To inline these invariant constants, select **Inline parameters**.

Note If your model contains Model blocks and the top model has **Inline parameters** selected, then **Inline parameters** must be selected for all referenced models. If a referenced model does not have **Inline parameters** selected, the Simulink engine temporarily enables this option while generating code for the referenced model, then disables it again when the build completes. Thus the referenced model is left in its previous state and you do not need to resave. For the top model, **Inline parameters** can be either selected or cleared.

Execution Speed

- “Inline Parameters” on page 21-2
- “Configure Loop Unrolling Threshold” on page 21-4
- “Optimize Code Generated for Vector Assignments” on page 21-6
- “Generate Target Optimizations Within Algorithm Code” on page 21-10

Inline Parameters

When you select the **Inline parameters** configuration parameter:

- The Simulink Coder code generator uses the numerical values of model parameters, instead of their symbolic names, in generated code.

  If the value of a parameter is a workspace variable, or an expression including one or more workspace variables, the variable or expression is evaluated at code generation time. The hard-coded result value appears in the generated code. The inlined parameter is a constant in the generated code, and is no longer tunable. That is, it is not visible to externally written code, and its value cannot be changed at runtime.
- The **Configure** button becomes enabled. Clicking the **Configure** button opens the Model Parameter Configuration dialog box.

  The Model Parameter Configuration dialog box lets you remove individual parameters from inlining and declare them to be tunable variables (or global constants). When you declare a parameter tunable, the Simulink Coder product generates a storage declaration that allows the parameter to be interfaced to externally written code. This enables your hand-written code to change the value of the parameter at run-time.

  The Model Parameter Configuration dialog box lets you improve overall efficiency by inlining most parameters, while at the same time retaining the flexibility of run-time tuning for selected parameters.

  See “Parameters” on page 8-11 for more information on interfacing parameters to externally written code.

Inline parameters also instructs the Simulink engine to propagate constant sample times. The engine computes the output signals of blocks that have constant sample times once during model startup. This improves execution speed because such blocks do not compute their outputs at every time step of the model.

You can select the **Inline invariant signals** code generation option (which also places constant values in generated code) only when **Inline parameters** is selected. See “Inline Invariant Signals” on page 20-14.

Referenced Models

When a top model uses referenced models or if a model is referenced by another model:

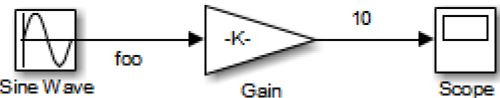
- For all referenced models, select **Inline parameters** if the top model has **Inline parameters** selected.
- For the top model, you can either select or clear **Inline parameters**.

When the top model specifies **Inline parameters** to be `on`, you cannot use the Model Parameter Configuration dialog box to tune parameters that are passed to referenced models. To tune such parameters, you must declare them in the referenced model's workspace, and then pass run-time values (or expressions) for them in argument lists specified for each Model block that references that model. For more information, see “Using Model Arguments”.

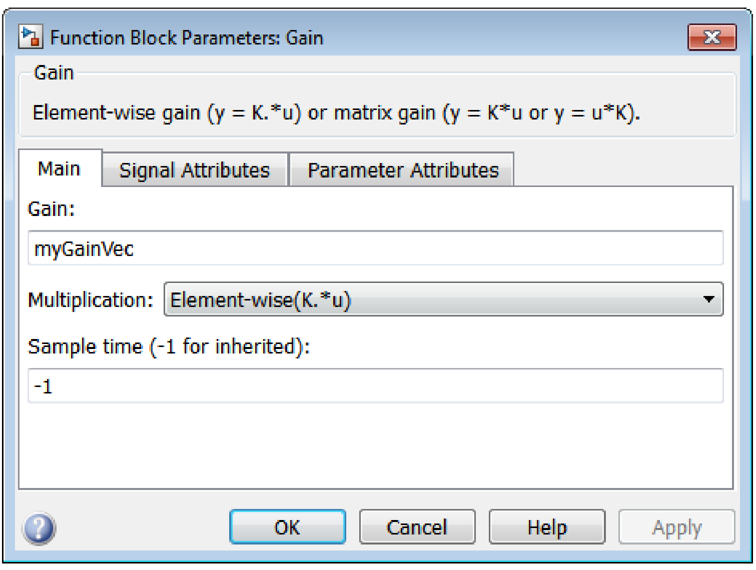
Configure Loop Unrolling Threshold

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.

Optimize Code Generated for Vector Assignments

In this section...

"Configure Model to Optimize Code Generated for Vector Assignments" on page 21-6

"Optimize Code Generated for Vector Assignments Using memcpy" on page 21-7

Configure Model to Optimize Code Generated for Vector Assignments

The **Use memcpy for vector assignment** option lets you optimize generated code for vector assignments by replacing `for` loops with `memcpy` function calls. The `memcpy` function can be more efficient than `for`-loop controlled element assignment for large data sets. Where `memcpy` offers improved execution speed, you can also use this model option to specify that the generated code use `memcpy` when assigning a vector signal.

Selecting the **Use memcpy for vector assignment** option 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, 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 `for` loops are replaced with `memcpy` function calls. 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.

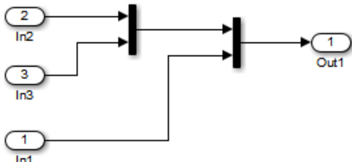
Note: If you are licensed for Embedded Coder software, you can use a code replacement library (CRL) to provide your own custom implementation of the `memcpy` function to be used in generated model code. For more information, see "Memory Function Code Replacement".

Optimize Code Generated for Vector Assignments Using memcpy

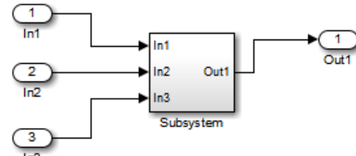
To examine the result of using the **Use memcpy for vector assignment** option on the generated vector assignment code, perform the following steps:

1 Create a model that generates signal vector assignments. For example,

a Use In, Out, and Mux blocks to create the following model.



b Select the diagram and use **Edit** > **Subsystem** to make it a subsystem.



c Open Model Explorer and configure the **Signal Attributes** for the `In1`, `In2`, and `In3` source blocks. For each, set **Port dimensions** to `[1,100]`, and set **Data type** to `int32`. Apply the changes.

d Go to the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box and clear the **Use memcpy for vector assignment** option. Apply the changes and save the model. In this example, the model is saved to the name `vectorassign`.

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 on the `vectorassign.c` section and inspect the model output function. Notice that the vector assignments are implemented using `for` loops.

```
/* Model output function */
static void vectorassign_output(void)
{
   int32_T i;

   /* Outport: '<Root>/Out1' incorporates:
    *  Inport: '<Root>/In1'
    *  Inport: '<Root>/In2'
    *  Inport: '<Root>/In3'
    */
   for (i = 0; i < 100; i++) {
      vectorassign_Y.Out1[i] = vectorassign_U.In2[i];
   }

   for (i = 0; i < 100; i++) {
      vectorassign_Y.Out1[i + 100] = vectorassign_U.In3[i];
   }

   for (i = 0; i < 100; i++) {
      vectorassign_Y.Out1[i + 200] = vectorassign_U.In1[i];
   }

   /* End of Outport: '<Root>/Out1' */
```

4 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.

5 In the HTML code generation report, click on the `vectorassign.c` section and inspect the model output function. Notice that the vector assignments now are implemented using `memcpy` function calls.

```
/* Model output function */
static void vectorassign_output(void)
{
   int32_T i;

   /* Outport: '<Root>/Out1' incorporates:
    *  Inport: '<Root>/In1'
    *  Inport: '<Root>/In2'
    *  Inport: '<Root>/In3'
    */
   memcpy(&vectorassign_Y.Out1[0], &vectorassign_U.In2[0], 100U * sizeof(int32_T));
   memcpy(&vectorassign_Y.Out1[100], &vectorassign_U.In3[0], 100U * sizeof(int32_T));
   memcpy(&vectorassign_Y.Out1[200], &vectorassign_U.In1[0], 100U * sizeof(int32_T));
 }
```

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	*EC only*—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	• `rtwdemo_crl_script` • "What Is Code Replacement?" and "What Is Code Replacement Customization?" in the Embedded Coder documentation
You want to control how code generation technology declares, stores, and represents signals, tunable parameters, block states, and data objects in generated code	*EC only*—Design (create) and apply custom storage classes	• `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.

Memory Usage

- "Minimize Memory Requirements During Code Generation" on page 22-2
- "Implement Logic Signals as Boolean Data" on page 22-3
- "Reduce Memory Requirements for Signals" on page 22-4
- "Reuse Memory Allocated for Signals" on page 22-5
- "Customize Stack Space Allocation" on page 22-6

Minimize Memory Requirements During Code Generation

When the Simulink Coder product generates code, it creates an intermediate 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 intermediate 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.

Implement Logic Signals as Boolean Data

When you select the **Implement logic signals as Boolean data (vs. double)** check box, blocks that generate logic signals output Boolean signals. When you clear this check box, blocks that generate logic signals output double signals.

The check box is selected by default because it reduces memory requirements (a Boolean signal typically requires one byte in memory while a double signal requires eight bytes in memory). Clear this check box for compatibility with models created using earlier versions of Simulink that support only double signals. For details about this check box, see "Implement logic signals as Boolean data (vs. double) ".

Reduce Memory Requirements for Signals

To reuse signal memory, which can reduce memory requirements of your real-time program, select the configuration parameter **Signal storage reuse**. Disabling **Signal storage reuse** makes all block outputs global and unique, which in many cases significantly increases RAM and ROM usage.

For more details on the **Signal storage reuse** option, see “Signals” on page 8-46.

Note Selecting **Signal storage reuse** also enables the **Enable local block outputs**, **Reuse block outputs**, **Eliminate superfluous local variables (expression folding)**, and **Minimize data copies between local and global variables** options on the **Optimization** > **Signals and Parameters** pane. See “Declare Signals as Local Function Data” on page 20-13 and “Reuse Memory Allocated for Signals” on page 22-5.

Reuse Memory Allocated for Signals

Two parameters provide the capability to reuse memory allocated for signals: **Reuse block output** and **Minimize data copies between local and global variables**.

Selecting **Reuse block output** directs the code generator to reuse signal memory. When **Reuse block output** is cleared, signals are stored in unique locations.

Reuse block output is enabled only when you select **Signal storage reuse**.

Selecting **Minimize data copies between local and global variables** reuses existing global variables to store temporary results. Clearing **Minimize data copies between local and global variables** writes data for block outputs to local variables. You must select **Signal storage reuse** to enable **Minimize data copies between local and global variables**.

See "Signals" on page 8-46 for more information (including generated code example) on **Reuse block output**, **Minimize data copies between local and global variables**, and other signal storage options.

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 Simulink Coder software (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".

Verification

23

Simulation and Code Comparison

- “Configure Signal Data for Logging” on page 23-2
- “Log Simulation Data” on page 23-4
- “Run Executable and Load Data” on page 23-6
- “Visualize and Compare Results” on page 23-7

Configure Signal Data for Logging

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 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:

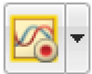
   - Specify the **Format** parameter as `Structure with time`.
   - Clear the **States** parameter check box.
   - Select the **Signal logging** parameter.
   - Select the **Record logged workspace data in Simulation Data Inspector** parameter to enable logged signal data to send to the Simulation Data Inspector after the simulation is finished.
8. Save the model.

Proceed to "Log Simulation Data" on page 23-4.

Log Simulation Data

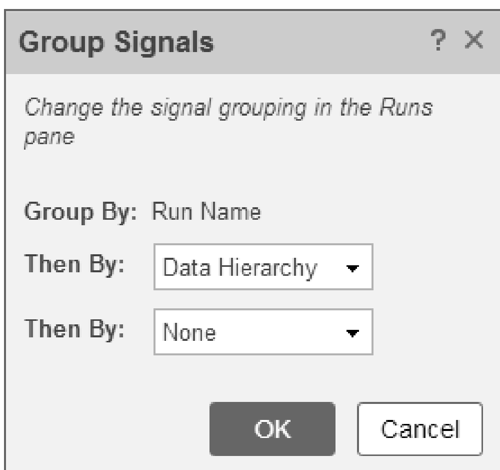
Run the simulation, log the signal data, and view the data in the Simulation Data Inspector.

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.

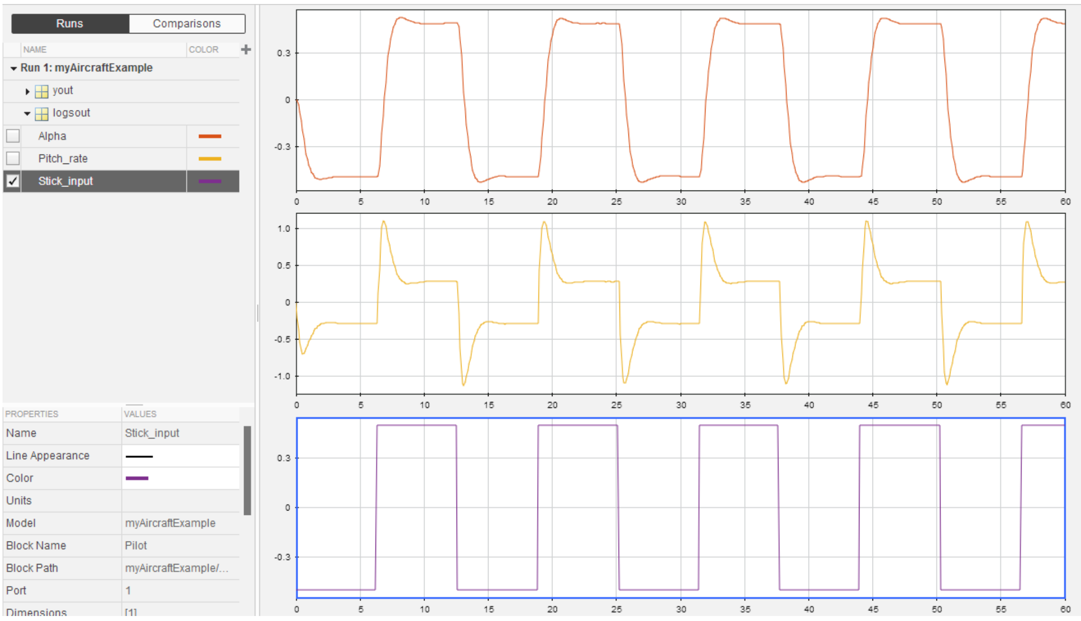
   c Click **OK**.
4 Click the `logsout` expander to view the logged signals.
5 Click the **Format** tab.
6 Click the **Subplots** button and select `3x1` to show three subplots.
7 For each signal:

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 23-6.

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** menu 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
   ```

Proceed to “Visualize and Compare Results” on page 23-7.

Visualize and Compare Results

When you follow the example sequence that began in "Configure Signal Data for Logging" on page 23-2, 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.

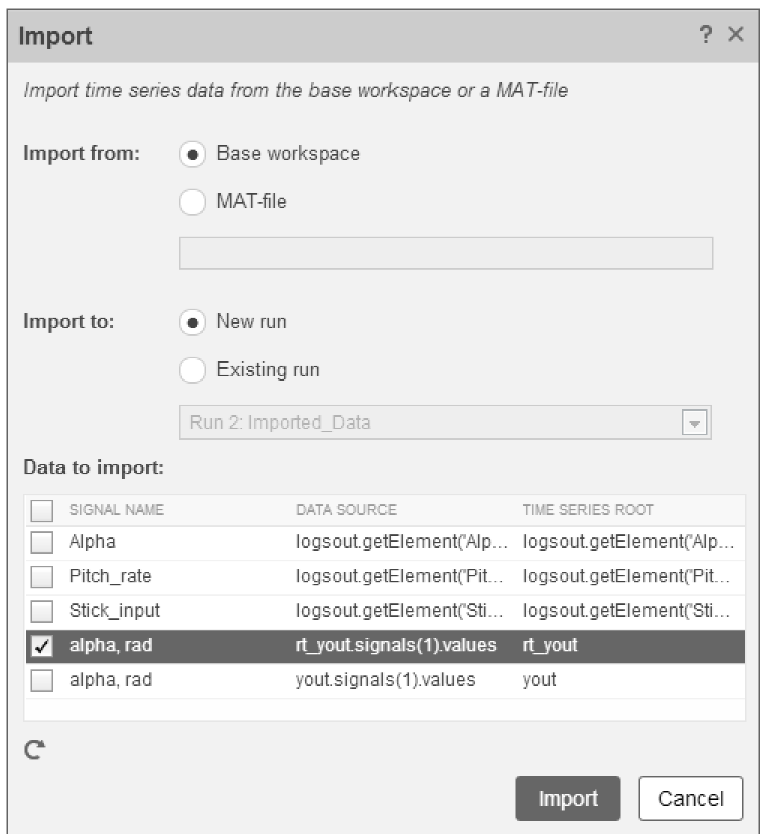
   b Specify **Import from** as **Base workspace**.

   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**.



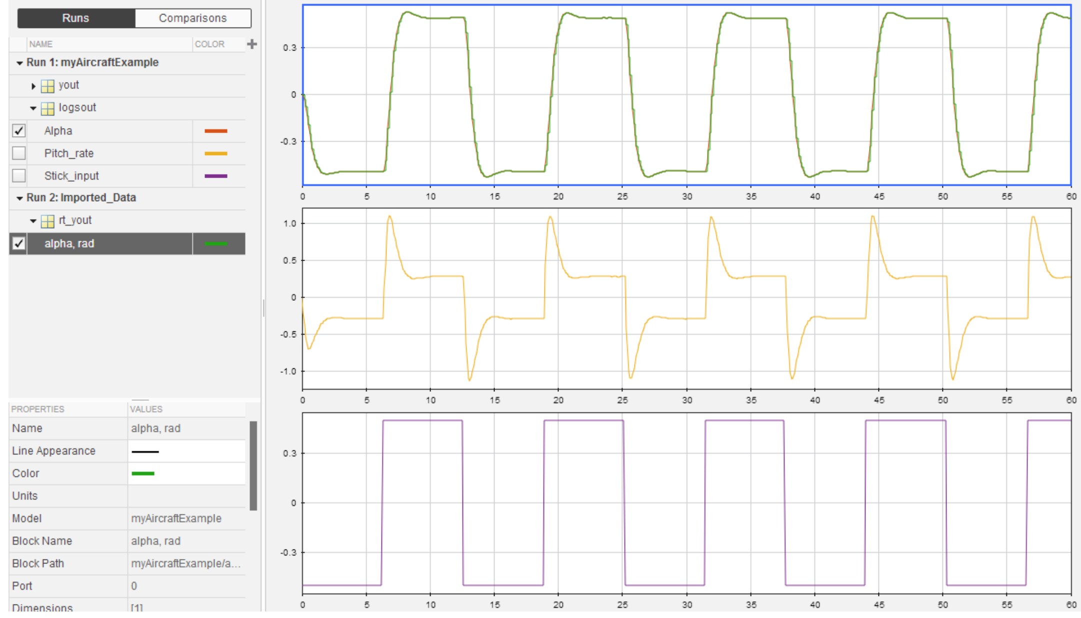
The selected data is now under **Run 2: Imported_Data**.

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.

Customization

Build Process Integration

- “Control Build Process Compiling and Linking” on page 24-2
- “Cross-Compile Code Generated on Microsoft Windows” on page 24-4
- “Control Library Location and Naming During Build” on page 24-7
- “Recompile Precompiled Libraries” on page 24-13
- “Customize Post-Code-Generation Build Processing” on page 24-14
- “Configure Generated Code with TLC” on page 24-19
- “Customize Build Process with STF_make_rtw_hook File” on page 24-21
- “Customize Build Process with sl_customization.m” on page 24-27
- “Replace the STF_rtw_info_hook Mechanism” on page 24-32
- “Customize Build to Use Shared Utility Code” on page 24-33

Control Build Process Compiling and Linking

After generating code for a model, the Simulink Coder 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 Simulink Coder software generates code for the model, including a makefile.
- **Generate makefile** option

  When you clear this option, the Simulink Coder software does not generate 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 24-14.
- 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<|. By default, the Simulink Coder software expands the |>TGT_FCN_LIB<| token to indicate the default math library used in generated code. You can use this token in a makefile conditional statement to specify compiler options. Possible |>TGT_FCN_LIB<| token values are:

Value	Generates Calls To
C89/C90 (ANSI)	ISO®/IEC 9899:1990 C standard math library for floating-point functions
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 Simulink Coder software 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 `*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`).

Control Library Location and Naming During Build

In this section...

"Library Control Parameters" on page 24-7
"Specify the Location of Precompiled Libraries" on page 24-9
"Control the Location of Model Reference Libraries" on page 24-10
"Control the Suffix Applied to Library File Names" on page 24-11

Library Control Parameters

Two configuration parameters, `TargetPreCompLibLocation` and `TargetLibSuffix`, are available for you to use to control values placed in Simulink Coder generated makefiles during the token expansion from template makefiles (TMFs). You can use these parameters to

- Specify the location of precompiled libraries, such as blockset libraries or the Simulink Coder 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',...
    '_diab.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\precomplibs');
```

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 24-10.

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 provided by TargetLibSuffix 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, as follows:

`modelname+suffix.extension_from_TargetLibSuffix`

With the toolchain approach, the final binary name is composed of modelname, the suffix, and the extension provided by the toolchain approach, as follows:

`model+suffix.extension_from_toolchain_approach`

The extension provided by the toolchain approach comes from the file extension of the static library created by the build tool. To get this information:

Create the toolchain object. For example, enter: `tc = ToolchainInfo used in the model`

Then get the build tool name. For example, enter:

```
tool = tc.getBuildTool('C Compiler');
```
or

```
tool = tc.getBuildTool('C++ Compiler');
```
And get the extension. For example, enter:

```
extension_from_ToolchainInfo = tool.getFileExtension('Static Library')
```

Note: Note: If you do not set the TargetLibSuffix parameter is not set, template makefile and toolchain approaches behave the same. See "Customize Library File Suffix and File Type".

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 16-103 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 library, the location specified in `rtwmakecfg.m` is typically a location specific to the blockset or the Simulink Coder product. It is expected that the library will exist in this location and it is linked against during Simulink Coder 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 library is expected to be created using the target-specific cross-compiler and placed in the target-specific location for use during the Simulink Coder build process. Libraries intended to be supported by the target should be compiled and placed in the target-specific location.

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 the TMF-to-makefile conversion, the Simulink Coder build process replaces the token `|>EXPAND_LIBRARY_LOCATION<|` with the specified location in the

rtwmakecfg.m file. 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_diab.library
```

By default, TargetPreCompLibLocation is an empty string and the Simulink Coder build process uses the location specified 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 Simulink Coder build process by default:

- Copies libraries used by the referenced models to the parent model's build folder
- Assigns 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 Simulink Coder build process assigns the library name sub_rtwlib.lib to MODELREF_LINK_LIBS and 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 Simulink Coder build process:

- Does not copy libraries used by the referenced models to the parent model's build folder
- Assigns the relative paths and file names of the libraries to MODELREF_LINK_LIBS in the generated makefile

When using this technique, the Simulink Coder build process assigns a relative path such as ../slprj/grt/sub/sub_rtwlib.lib to MODELREF_LINK_LIBS and uses the path to gain access to the library file at link time.

To change to the non-default behavior on platforms other than the Macintosh platform, enter the following command in the **Make command** field of the **Code Generation** pane of the Configuration Parameters dialog box:

```
make_rtw USE_MDLREF_LIBPATHS=1
```

If you specify other Make command arguments, such as `OPTS="-g"`, the order that 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 string 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 16-103.

  In this case, a target can set the parameter in its select callback. For example:

  ```
  slConfigUISetVal(hDlg, hSrc, 'TargetLibSuffix',...
  '_diab.library');
  ```

  Alternatively, you can use a call to the `set_param` function. For example:

  ```
  set_param(model,'TargetLibSuffix','_diab.library');
  ```

  During the TMF-to-makefile conversion, the Simulink Coder 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_diab.library
  ```

By default, TargetLibSuffix is set to an empty string. In this case, the Simulink Coder build process replaces the token |>EXPAND_LIBRARY_SUFFIX<| with an empty string.

- 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 Simulink Coder 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_diab.library
MODELLIB            = topmodellib_diab.library
MODELREF_LINK_LIBS = \
refmodel1_rtwlib_diab.library refmodel2_rtwlib_diab.library
```

By default, the TargetLibSuffix parameter is an empty string. In this case, the Simulink Coder 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 would be:

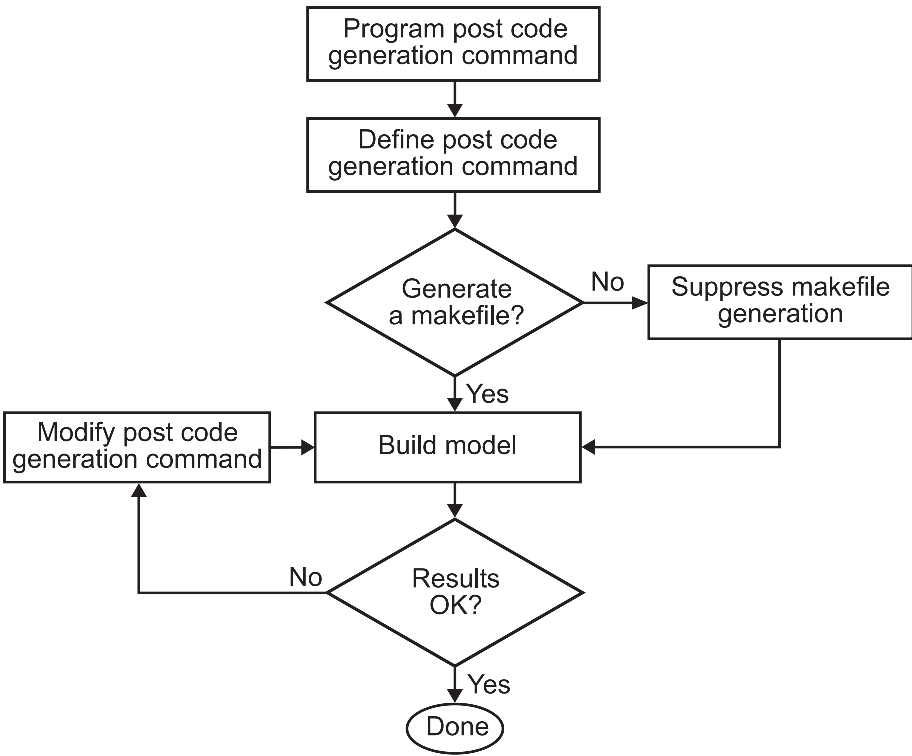
```
SHARED_LIB          = ..\slprj\ert\_sharedutils\rtwshared.lib
MODELLIB            = topmodellib.lib
MODELREF_LINK_LIBS = refmodel1_rtwlib.lib refmodel2_rtwlib.lib
```

Recompile Precompiled Libraries

You can recompile precompiled libraries included as part of the Simulink Coder product, 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 16-108.

Customize Post-Code-Generation Build Processing

The Simulink Coder product 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. The next figure and the steps that follow show the general workflow for setting up such 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 Simulink Coder 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”. “Program a Post Code Generation Command” on page 24-15 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 Simulink Coder 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", 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 analyzegencode(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 Simulink Coder 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 Simulink Coder 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 `analyzegencode`.

```
set_param(model, 'PostCodeGenCommand',...
'analyzegencode(buildInfo)');
```

Suppress Makefile Generation

The Simulink Coder product 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 Simulink Coder build process to not generate 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 24-15 and "Define a Post Code Generation Command" on page 24-16.

Configure Generated Code with TLC

In this section...
“About Configuring Generated Code with TLC” on page 24-19
“Assigning Target Language Compiler Variables” on page 24-19
“Set Target Language Compiler Options” on page 24-20

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
	are declared by block outputs in the model cannot exceed `MaxStackSize` (in bytes). `MaxStackSize` 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 `MaxStackSize` is `rtInf`, that is, unlimited stack size. **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 `MaxStackSize` 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.
`MaxStackVariableSize=`*`N`*	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 `MaxStackVariableSize` is allocated in global, rather than local, memory. The default is 4096.
`WarnNonSaturatedBlocks=`*`value`*	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: • `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” and “Configure TLC”.

Customize Build Process with STF_make_rtw_hook File

In this section...

"The STF_make_rtw_hook File" on page 24-21
"Conventions for Using the STF_make_rtw_hook File" on page 24-21
"STF_make_rtw_hook.m Function Prototype and Arguments" on page 24-22
"Applications for STF_make_rtw_hook.m" on page 24-24
"Control Code Regeneration Using STF_make_rtw_hook.m" on page 24-25
"Use STF_make_rtw_hook.m for Your Build Procedure" on page 24-26

The 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.

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 automatically calls `STF_make_rtw_hook`, passing in the`hookMethod` argument and other arguments. You implement only those hook methods that your build process requires.

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 string `_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 24-22.

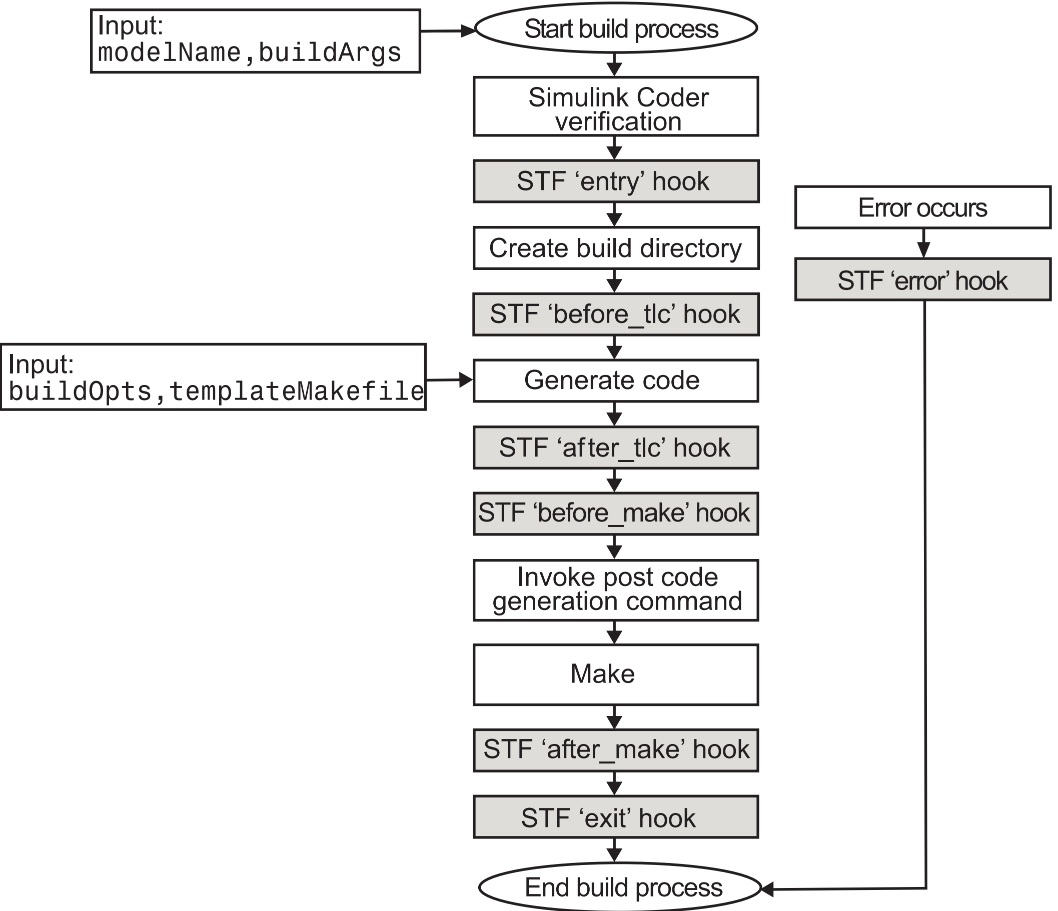
STF_make_rtw_hook.m Function Prototype and Arguments

The function prototype for `STF_make_rtw_hook` is:

```
function STF_make_rtw_hook(hookMethod, modelName, rtwRoot, templateMakefile,
buildOpts, buildArgs)
```

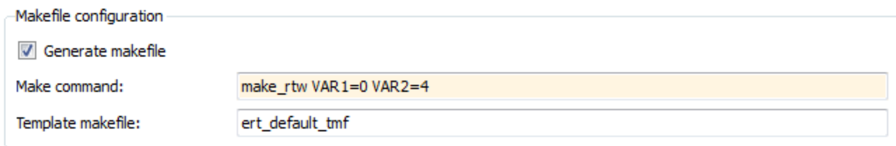
The arguments are defined as:

- `hookMethod`: String 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`: String 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 array specifying a list of additional files to be compiled.

  - codeFormat: Character array specifying the code format for the target. (ERT-based targets must use the 'Embedded-C' code format.)
  - noninlinedSFcns: Cell array specifying a list of noninlined S-functions in the model.
  - compilerEnvVal: String specifying the compiler environment variable value (for example, C:\Applications\Microsoft Visual).
- buildArgs: Character array containing the argument to make_rtw. When you invoke the build process, buildArgs is copied from the argument string following "make_rtw" in the **Make command** field on the **Code Generation** pane of the Configuration Parameters dialog box.



For example, the make arguments from the **Make command** field in the preceding figure generate the following:

```
% make -f untitled.mk VAR1=0 VAR2=4
```

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 string 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 17-29.) 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 (located in the *matlabroot*/toolbox/coder/embeddedcoder folder) 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 24-21. 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

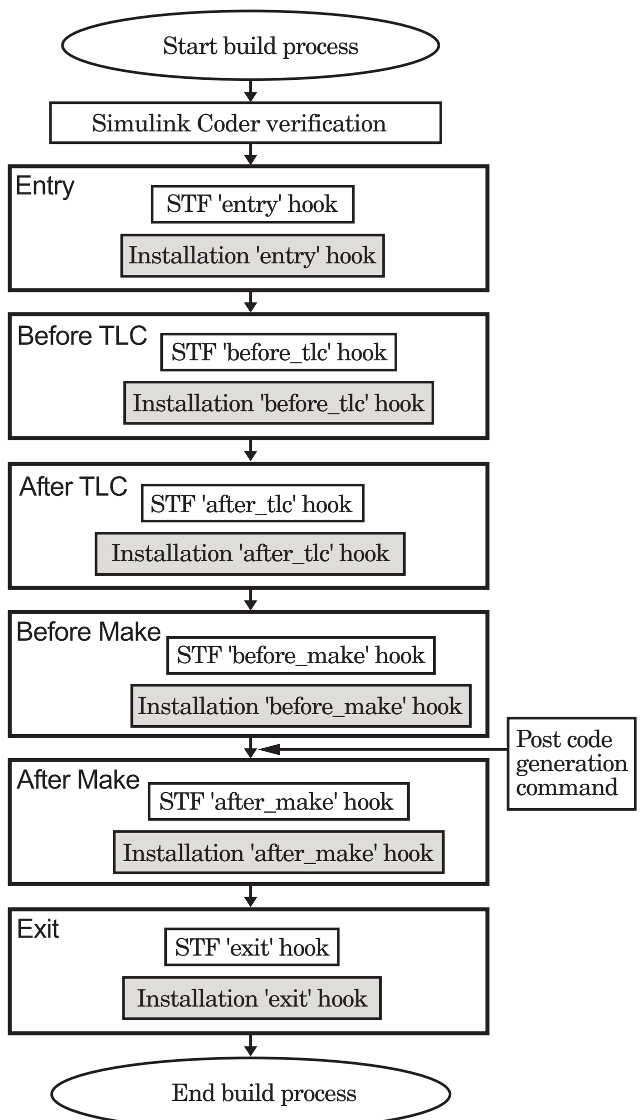
In this section...
"The sl_customization.m File" on page 24-27
"Register Build Process Hook Functions Using sl_customization.m" on page 24-29
"Variables Available for sl_customization.m Hook Functions" on page 24-29
"Example Build Process Customization Using sl_customization.m" on page 24-30

The sl_customization.m File

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".

The `sl_customization.m` file can be used to register installation-specific hook functions to be invoked during the Simulink Coder 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 24-21) and post-code generation commands (described in "Customize Post-Code-Generation Build Processing" on page 24-14).

The following figure shows the relationship between installation-level hooks and the other available mechanisms for customizing the build process.


Start build process
Simulink Coder verification
Entry
STF 'entry' hook
Installation 'entry' hook
Before TLC
STF 'before_tlc' hook
Installation 'before_tlc' hook
After TLC
STF 'after_tlc' hook
Installation 'after_tlc' hook
Before Make
STF 'before_make' hook
Installation 'before_make' hook
Post code generation command
After Make
STF 'after_make' hook
Installation 'after_make' hook
Exit
STF 'exit' hook
Installation 'exit' hook
End build process

Register Build Process Hook Functions Using sl_customization.m

To register installation-level hook functions that will be invoked during the Simulink Coder 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 Simulink Coder 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 Simulink Coder 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 Simulink Coder build. For the hook function source code, see Example 2: CustomRTWEntryHook.m and Example 3: CustomRTWPostProcessHook.m.

Example 1: sl_customization.m for Simulink Coder 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 Simulink Coder builds. For example, if you open the ERT-based model `rtwdemo_udt`, open the **Code Generation** pane of the Configuration Parameters dialog box, and click the **Build** button to initiate a Simulink Coder 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 the STF_rtw_info_hook Mechanism

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 “Target” on page 10-12 .

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 "Logging" on page 17-99 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 26-78 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/`
  - `grt_lcc.tmf`
  - `grt_vc.tmf`
  - `grt_unix.tmf`
- ERT: *`matlabroot`*`/rtw/c/ert/`
  - `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

- "Customize an ASAP2 File" on page 25-2
- "Create a Transport Layer for External Communication" on page 25-8

Customize an ASAP2 File

In this section...

"About ASAP2 File Customization" on page 25-2
"ASAP2 File Structure on the MATLAB Path" on page 25-2
"Customize the Contents of the ASAP2 File" on page 25-3
"ASAP2 Templates" on page 25-4
"Customize Computation Method Names" on page 25-6
"Suppress Computation Methods for FIX_AXIS" on page 25-7

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 folder contains TLC files that generate ASAP2 files, asamlib.tlc, asap2lib.tlc, asap2main.tlc, and asap2grouplib.tlc. These files are included by the selected **System target file**. (See "Targets Supporting ASAP2" on page 17-159.)
- ASAP2 target files

  The *matlabroot*/toolbox/rtw/targets/asap2/asap2 folder contains the ASAP2 system target file and other control files.
- Customizable TLC files

  The *matlabroot*/toolbox/rtw/targets/asap2/asap2/user 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 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 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 25-4.
   - 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 25-4.
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 `DocUnits` 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 25-8
"Design of External Mode" on page 25-8
"External Mode Communications Overview" on page 25-11
"External Mode Source Files" on page 25-12
"Implement a Custom Transport Layer" on page 25-17

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 RTWin 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 Real-Time Windows Target 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)	• *matlabroot*/toolbox/coder/simulinkcoder_core/ext_mode/host/common/rtiostream_interface.c • *matlabroot*/rtw/c/src/rtiostream/rtiostreamtcpip/-rtiostream_tcpip.c
	Server (target)	• *matlabroot*/rtw/c/src/ext_mode/common/rtiostream_interface.c • *matlabroot*/rtw/c/src/rtiostream/rtiostreamtcpip/-rtiostream_tcpip.c
Serial	Client (host)	• *matlabroot*/toolbox/coder/simulinkcoder_core/ext_mode/host/serial/ext_serial_transport.c • *matlabroot*/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c
	Server (target)	• *matlabroot*/rtw/c/src/ext_mode/serial/ext_svr_serial_transport.c • *matlabroot*/rtw/c/src/rtiostream/rtiostreamserial/rtiostream_serial.c

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 25-17.

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 25-20.

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 25-13

- "Server (Target) Source Files" on page 25-15
- "Other Files in the Server Folder" on page 25-16

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, 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/rtiostreamtserial/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` 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 part of the run-time interface and are linked into the *model*.exe executable. They are located within *matlabroot*/rtw/c/src/ext_mode/ 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/rtiostreamtserial/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 25-17
- "Create a Custom Client (Host) Transport Protocol" on page 25-17
- "Register a Custom Client (Host) Transport Protocol" on page 25-19
- "Create a Custom Server (Target) Transport Protocol" on page 25-20
- "Serial Receive Buffer Smaller than 64 Bytes" on page 25-22

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 the table MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files 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 25-19.

The following table lists the commands for building the standard `ext_comm` module on PC and UNIX platforms, and for building the standard `ext_serial_win32` model on a PC platform.

MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files

Platform	Commands

PC, TCP/IP	`>> 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 ...`
	`-lmwrtiostreamutils ...`
	`-DEXTMODE_TCPIP_TRANSPORT ...`
	`-DSL_EXT_DLL -output toolbox\coder\simulinkcoder_core\ext_comm`
	
	Note: The `rtiostream_interface.c` function defines RTIOSTREAM_SHARED_LIB as `libmwrtiostreamtcpip` and dynamically loads the MathWorks TCP/IP `rtiostream` shared library. Modify this file if you need to load a different `rtiostream` shared library.

UNIX, TCP/IP	Use the PC commands but change `-DSL_EXT_DLL` to `-DSL_EXT_SO`, and replace back slashes with forward slashes.
Mac, TCP/IP	Use the PC commands but change `-DSL_EXT_DLL` to `-DSL_EXT_DYLIB`, and replace back slashes with forward slashes.
PC, serial	`>> cd (matlabroot)`

<table>
<tr><th>Platform</th><th>Commands</th></tr>
<tr><td></td><td><pre>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 ...
-DEXTMODE_SERIAL_TRANSPORT -DSL_EXT_DLL ...
-output toolbox\coder\simulinkcoder_core\ext_serial_win32_comm</pre>
Note: The <code>rtiostream_interface.c</code> function defines RTIOSTREAM_SHARED_LIB 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.</td></tr>
<tr><td>UNIX, serial</td><td>Use the PC commands but change <code>-DSL_EXT_DLL</code> to <code>-DSL_EXT_SO</code>, and replace back slashes with forward slashes.</td></tr>
<tr><td>Mac, serial</td><td>Use the PC commands but change <code>-DSL_EXT_DLL</code> to <code>-DSL_EXT_DYLIB</code>, and replace back slashes with forward slashes.</td></tr>
</table>

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

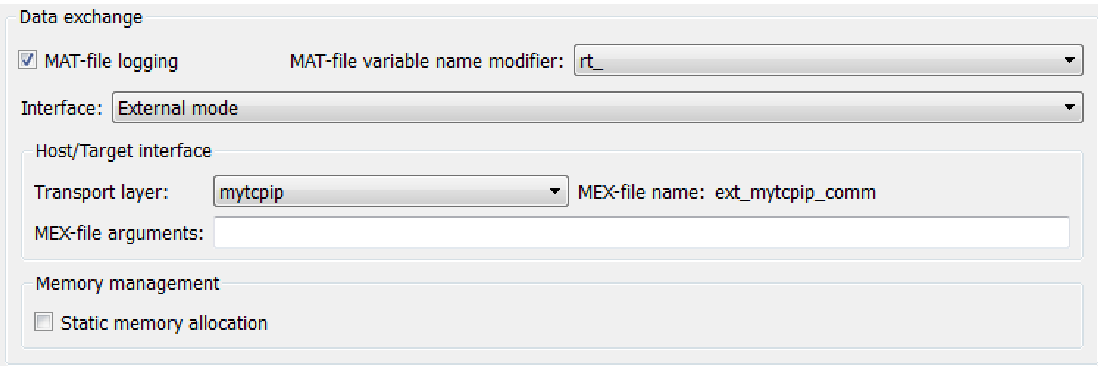
- *`stf`*`.tlc` is the name of the system target file for which the transport will be registered (for example, `'grt.tlc'`)

- *`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.



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”, “Customize Build Process with STF_make_rtw_hook File”, and “Customize Build Process with sl_customization.m”.) 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.

Custom Target Development

- "About Embedded Target Development" on page 26-2
- "Sample Custom Targets" on page 26-8
- "Target Development Mechanics" on page 26-10
- "Customize System Target Files" on page 26-26
- "Customize Template Makefiles" on page 26-56
- "Support Optional Features" on page 26-77
- "Interface to Development Tools" on page 26-96
- "Device Drivers and Target Preferences" on page 26-106

About Embedded Target Development

In this section...
“Custom Targets” on page 26-2
“Types of Targets” on page 26-2
“Recommended Features for Embedded Targets” on page 26-4

Custom Targets

The targets bundled with the Simulink Coder product are suitable for many different applications and development environments. Third-party targets provide additional versatility. However, 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 Simulink Coder product 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

- “Introduction” on page 26-2
- “Rapid Prototyping Targets” on page 26-3
- “Turnkey Production Targets” on page 26-3
- “Verifying Targets With SIL and PIL Testing” on page 26-4
- “HIL Simulation Targets” on page 26-4

Introduction

The following sections describe several types of targets intended for different use cases. There is a progression of capabilities from the first (baseline or rapid prototyping)

to second (turnkey 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 26-4, 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 Simulink Coder 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 Simulink Coder software.

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 26-8.

For more information on SIL and PIL, see “About 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 26-4
- “Integration with Target Development Environments” on page 26-5
- “Observing Execution of Target Code” on page 26-6
- “Deployment and Hardware Issues” on page 26-6

Basic Target Features

- You can base targets on the Simulink Coder 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 that target's Embedded-C code format, and should inherit the options defined in the ERT target's system target file. By following these recommendations, your target has the production code generation capabilities of the ERT target.

See “Customize System Target Files” on page 26-26 for further details on the inheritance mechanism, setting the code format, 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 the Simulink Coder concepts of single-tasking and multitasking solver modes for model execution. Tasking support comes almost “for free” with the ERT target, but you should thoroughly understand how it works before implementing an ERT-based target.

  Implementation of timer interrupt-based execution is documented in “Scheduling ” and “Time-Based Scheduling”.
- 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 “Standalone Programs (No Operating System)” for information on generated and static main program modules.
- Follow the guidelines in “Folder and File Naming Conventions” on page 26-10.

Integration with Target Development Environments

- Most cross-development systems run under a Microsoft Windows PC host. Your target should support the Windows XP 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 most common approach is to use a template makefile to generate standard

makefiles with the make utility provided with your development environment. "Customize Template Makefiles" on page 26-56 describes the structure of template makefiles.

Another approach with IDE-based tools is project file creation and/or Microsoft Windows Component Object Model (COM) automation.

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 26-96 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. See the Web page example here "Creating a Communications Channel for Target Connectivity". 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 Embedded Coder product.

  Another option is to support Simulink External mode over a serial interface (RS-232). See the "Host/Target Communication" 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 “Integrate Device Drivers” on page 26-106.

- 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 26-19 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 26-96.
- 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.

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.

Target Development Mechanics

In this section...
"Folder and File Naming Conventions" on page 26-10
"Components of a Custom Target" on page 26-11
"Key Folders Under Target Root (mytarget)" on page 26-15
"Key Files in Target Folder (mytarget/mytarget)" on page 26-18
"Additional Files for Externally Developed Targets" on page 26-20
"Target Development and the Build Process" on page 26-21

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 26-11
- "Code Components" on page 26-11
- "Control Files" on page 26-13

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 26-10.

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 `.m` 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

A Simulink Coder 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 Simulink Coder Software 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 Simulink Coder software 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 Simulink Coder `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. “Standalone Programs (No Operating System)” 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 Simulink Coder 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 Simulink Coder APIs. See “Data Exchange” 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 click **Build** (or type **Ctrl+B**). At this point, the Simulink Coder 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 26-21.) 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 intermediate 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 26-26.

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 vs. generating a project file for your compiler).

The Embedded Coder software provides a large number of TMFs suitable for different types of host-based development systems. These TMFs are located in *`matlabroot`*`/rtw/c/ert`. The standard TMFs are described in the "Template Makefiles and Make Options" section of the Simulink Coder documentation.

The detailed structure of the TMF is described in "Customize Template Makefiles" on page 26-56.

Hook Files

Simulink Coder 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 26-10 describes these requirements.

Key Folders Under Target Root (mytarget)

- "Target Root Folder (mytarget)" on page 26-15
- "Target Folder (mytarget/mytarget)" on page 26-16
- "Target Block Folder (mytarget/blocks)" on page 26-16
- "Development Tools Folder (mytarget/dev_tool1, mytarget/dev_tool2)" on page 26-17
- "Target Source Code Folder (mytarget/src)" on page 26-18

Target Root Folder (mytarget)

This folder contains the key subfolders for the target (see "Folder and File Naming Conventions" on page 26-10). 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 26-18 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/slblocks.m

This file allows a group of blocks to be integrated into the Simulink Library and Simulink Library Browser.

Example slblocks.m File

```
function blkStruct = slblocks
```

```
% 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>

   <demosection>
      <label>Multirate model</label>
      <demoitem>
         <label>MYTARGET demo</label>
         <file>mytarget_overview.html</file>
         <callback>mytarget_model</callback>
      </demoitem>
   </demosection>

</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 26-18
- "mytarget.tlc" on page 26-18
- "mytarget.tmf" on page 26-19
- "mytarget_genfiles.tlc" on page 26-19
- "mytarget_main.c" on page 26-19
- "STF_make_rtw_hook.m" on page 26-19
- "STF_rtw_info_hook.m (obsolete)" on page 26-20
- "info.xml" on page 26-20
- "mytarget_overview.html" on page 26-20

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 the Embedded Coder software.

"Customize System Target Files" on page 26-26 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 26-56.

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 26-96.

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 26-42 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

"Standalone Programs (No Operating System)" 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 24-21 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".

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 26-17).

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 26-21

- “mytarget/mytarget/mytarget_setup.m” on page 26-21
- “mytarget/mytarget/doc” on page 26-21

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 26-22
- “Build Process Phases and Information Passing” on page 26-22
- “Additional Information Passing Techniques” on page 26-23

About the Build Process

To develop an embedded target, you need a thorough understanding of the Simulink Coder 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 “Program Builds”.

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 26-23, 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 Simulink Coder related 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 and makefile tokens in the `rtwoptions` data structure.
- The template makefile (TMF), which generates 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 26-18 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

Options on the Simulink Coder related 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 Simulink Coder options and their values available in the TLC phase.

See “System Target File Structure” on page 26-27 for further information.

makevariable Field in rtwoptions Structure

Similarly, Simulink Coder options can be associated with a template makefile token, specified 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 26-27 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 accidently 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 "Using the rtwmakecfg.m API to Customize Generated Makefiles" on page 26-70.

Customize System Target Files

In this section...

"Control Code Generation With the System Target File" on page 26-26
"System Target File Naming and Location Conventions" on page 26-27
"System Target File Structure" on page 26-27
"Define and Display Custom Target Options" on page 26-35
"Tips and Techniques for Customizing Your STF" on page 26-41
"Create a Custom Target Configuration" on page 26-44

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 code format to be generated
- 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

  Note: You can save a configuration set with custom options to a MAT-file. However, when you load the MAT-file, some custom options might not appear in the Configuration Parameters dialog box.

- A mechanism for inheriting options from another target (such as the Embedded Real-Time (ERT) target)

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.

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 Simulink Coder product. Most of these files are stored in the target-specific folders under `matlabroot/rtw/c`. Additional STFs are stored under `matlabroot/toolbox/rtw/targets`.

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 26-10.

System Target File Structure

- "Overview" on page 26-27
- "Header Comments" on page 26-29
- "TLC Configuration Variables" on page 26-30
- "TLC Program Entry Point and Related %includes" on page 26-31
- "RTW_OPTIONS Section" on page 26-32
- "rtwgensettings Structure" on page 26-32
- "Additional Code Generation Options" on page 26-34
- "Model Reference Considerations" on page 26-35

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
%% Inital 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 string 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 Simulink Coder software to detect STFs. When the System Target File Browser is opened, the Simulink Coder software scans the MATLAB path for TLC files that have formatted header comments. The comments contain the following directives:

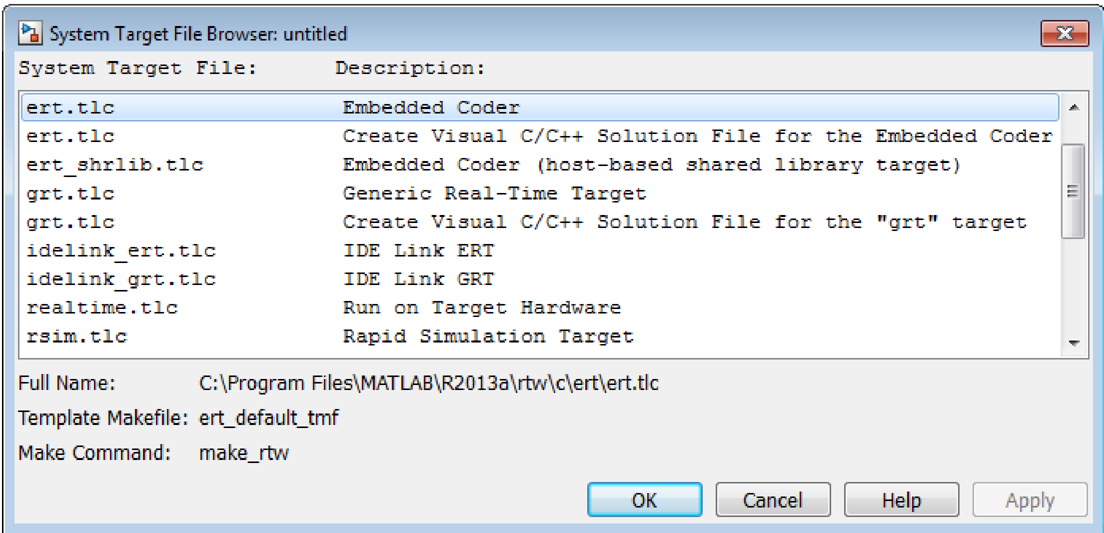
- SYSTLC: This string is a descriptor that appears in the browser.
- TMF: Name of the template makefile (TMF) to use during build process. When the target is selected, this filename is displayed in the **Template makefile** field of the **Code Generation** pane of the Configuration Parameters dialog 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 the Embedded Coder\
%%    TMF: RTW.MSVCBuild 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 (`.shn`) file. The figure below shows how these configurations appear in the System Target File Browser.



See “Create a Custom Target Configuration” on page 26-44 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 select the Embedded-C code format. Verify that values are assigned to the following variables:

- `CodeFormat`: The `CodeFormat` variable selects one of the available code formats. The `Embedded-C` format is used by the ERT target. Your ERT-based target should specify `Embedded-C` format. `Embedded-C` format is designed for production code, minimal memory usage, static memory allocation, and a simplified interface to generated code.

For information on other code formats, see “Targets and Code Formats”.

- `Language`: The only valid value is `C`, which enables support for `C` or `C++` code generation as specified by the configuration parameter `TargetLang` (see the `TargetLang` entry in “Parameter Command-Line Information Summary” in the Simulink Coder documentation for more information).
- `TargetType`: The Simulink Coder software 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 26-42 for an example of custom TLC code for execution after the main code generation entry point.

- "Target Development and the Build Process" on page 26-21 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 in the format specified by `CodeFormat`.

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 26-35.

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 26-32 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 string property defines the system target file from which options are to be inherited. See "Inheriting Target Options" on page 26-40.
- `rtwgensettings.SelectCallback`: this property 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 26-78.

- `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 string that identifies build folders created by the target. The build process appends the string 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" in the Simulink Coder documentation 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".)

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 26-78 for important information on STF and other modifications you may need to make to support the Simulink Coder model referencing features.

Define and Display Custom Target Options

- "Using rtwoptions to Display Custom Target Options" on page 26-35
- "Example System Target File With Customized rtwoptions" on page 26-39
- "Inheriting Target Options" on page 26-40

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 26-38 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.

Note: You can save a configuration set with custom options to a MAT-file. However, when you load the MAT-file, some custom options might not appear in the Configuration Parameters dialog box.

Example rtwoptions Structure

The following example is excerpted from *`matlabroot`*`/rtw/c/rtwsfcn/rtwsfcn.tlc`, the STF for the S-function target. 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        = 'S-Function Target';
rtwoptions(1).type          = 'Category';
rtwoptions(1).enable        = 'on';
rtwoptions(1).default       = 4;   % number of items under this category
                                   % excluding this one.
rtwoptions(1).popupstrings  = '';
rtwoptions(1).tlcvariable   = '';
rtwoptions(1).tooltip       = '';
rtwoptions(1).callback      = '';
rtwoptions(1).opencallback  = '';
rtwoptions(1).closecallback = '';
rtwoptions(1).makevariable  = '';

rtwoptions(2).prompt         = 'Create new model';
rtwoptions(2).type           = 'Checkbox';
rtwoptions(2).default        = 'on';
rtwoptions(2).tlcvariable    = 'CreateModel';
rtwoptions(2).makevariable   = 'CREATEMODEL';
rtwoptions(2).tooltip        = ...
  ['Create a new model containing the generated S-Function
  block inside it'];

rtwoptions(3).prompt         = 'Use value for tunable parameters';
rtwoptions(3).type           = 'Checkbox';
rtwoptions(3).default        = 'off';
rtwoptions(3).tlcvariable    = 'UseParamValues';
rtwoptions(3).makevariable   = 'USEPARAMVALUES';
rtwoptions(3).tooltip        = ...
  ['Use value for variable instead of variable name in generated block mask
  edit fields'];
```

```
% Override the default setting for model name prefixing because
% the generated S-function is typically used in multiple models.
rtwoptions(4).default       = 'on';
rtwoptions(4).tlcvariable   = 'PrefixModelToSubsysFcnNames';

rtwoptions(5).prompt         = 'Include custom source code';
rtwoptions(5).type           = 'Checkbox';
rtwoptions(5).default        = 'off';
rtwoptions(5).tlcvariable    = 'AlwaysIncludeCustomSrc';
rtwoptions(5).tooltip        = ...
  ['Always include provided custom source code in the generated code'];
```

The first element adds the **S-Function Target** pane under **Code Generation** in the Configuration Parameters dialog box. The options defined in `rtwoptions(2)`, `rtwoptions(3)`, and `rtwoptions(5)` display.

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.

For further examples of target-specific `rtwoptions` definitions, see "Example System Target File With Customized rtwoptions" on page 26-39.

rtwoptions Structure Fields Summary lists the fields of the `rtwoptions` structure.

rtwoptions Structure Fields Summary

Field Name	Description
`callback`	For examples of callback usage, see "Example System Target File With Customized rtwoptions" on page 26-39.
`closecallback` (obsolete)	Do not use `closecallback`. Use `rtwgensettings.PostApplyCallback` instead (see "rtwgensettings Structure" on page 26-32). `closecallback` is ignored. For examples of callback usage, see "Example System Target File With Customized rtwoptions" on page 26-39.
`default`	Default value of the option (empty if the `type` is `Pushbutton`).

Field Name	Description
enable	Must be 'on' or 'off'. If 'on', the option is displayed as an enabled item; otherwise, as a disabled item.
makevariable	Template makefile token (if any) associated with the option. The makevariable is expanded during processing of the template makefile. See "Template Makefile Tokens" on page 26-57.
modelReferenceParameter-Check	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 26-83.
NonUI	Element that is not displayed, but is used to invoke a close or open callback. See "NonUI Elements" on page 26-38.
opencallback (obsolete)	Do not use opencallback. Use rtwgensettings.SelectCallback instead (see "rtwgensettings Structure" on page 26-32). For examples of callback usage, see "Example System Target File With Customized rtwoptions" on page 26-39.
popupstrings	If type is Popup, popupstrings defines the items in the menu. Items are delimited by the "|" (vertical bar) character. The 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 string 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. Only the `opencallback` and `closecallback` fields of a `NonUI` element have significance. See the next section, "Example System Target File With Customized rtwoptions" on page 26-39 for an example.

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:

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`: illunstrates 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` your working folder.
2. Open a model of your choice.

3. Open the Configuration Parameters dialog box or Model Explorer 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 string property `rtwgensettings.DerivedFrom` 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` property 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 Simulink Coder software 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 Simulink Coder software 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 26-41
- “Required and Recommended %includes” on page 26-41
- “Handling Aliases for Target Option Values” on page 26-42
- “Supporting Multiple Development Environments” on page 26-44

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 26-42.

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 24-21.

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 strings represent the same hardware device, invoke the following function:

```
result = RTW.isHWDeviceTypeEq(type1,type2)
```

where *type1* and *type2* are strings containing target device type values or aliases.

The `RTW.isHWDeviceTypeEq` function returns true if *type1* and *type2* are strings 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 string 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.isTflEq

To test if two code replacement library (CRL) strings represent the same CRL, invoke the following function:

```
result = RTW.isTflEq(name1,name2)
```

where *name1* and *name2* are strings containing CRL values or aliases.

The `RTW.isTflEq` function returns true if *name1* and *name2* are strings representing the same code replacement library. For example, the following call returns true:

```
RTW.isTflEq('GNU','GNU C99 extensions')
```

For a description of the `CodeReplacementLibrary` parameter, see “Code replacement library”.

RTW.resolveTflName

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.resolveTflName(name)
```

where *name* is a string containing a CRL value or alias.

The `RTW.resolveTflName` function returns the value of the referenced code replacement library. For example, the following calls both return `'GNU C99 extensions'`:

```
RTW.resolveTflName('GNU')
RTW.resolveTflName('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 vs. generating a project file for your compiler).

One approach to this requirement is to implement multiple STFs. 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 26-45

- “my_ert_target Overview” on page 26-45
- “Creating Target Folders” on page 26-47
- “Create ERT-Based STF” on page 26-48
- “Create ERT-Based TMF” on page 26-53
- “Create Test Model and S-Function” on page 26-53
- “Verify Target Operation” on page 26-54

Introduction

This tutorial can supplement the example target guides described in “Sample Custom Targets” on page 26-8. For an introduction and example files to examine, try the example targets first.

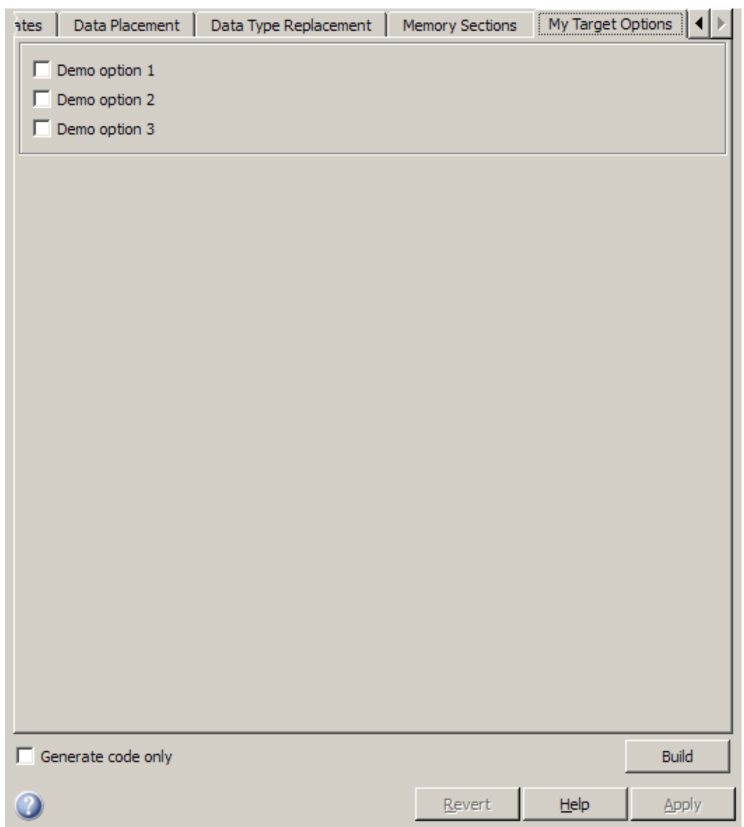
The purpose of this tutorial is to guide you through the process of creating an ERT-based target, `my_ert_target`. This exercise illustrates several tasks that are usually required when creating a custom target:

- 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 may 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 template makefile-based build, generating code and executables that run on the host system. `my_ert_target` uses the `lcc` compiler on a Microsoft Windows platform. This compiler was chosen because it is readily available and is distributed with the Simulink Coder product. On a Microsoft Windows 32-bit 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, check that you have a C compiler installed. You can then do this exercise by using Linux folder syntax.

You can test `my_ert_target` with a model that is compatible with the ERT target. (See "Code Generation Targets" in the Embedded Coder documentation.) Generated programs operate identically to ERT generated programs.

However, to simplify the testing of your target, we recommend testing with `targetmodel`, a very simple fixed-step model (see "Create Test Model and S-Function" on page 26-53). The S-Function block in `targetmodel` uses the source code from the `timestwo` example, and generates fully inlined code. See "S-Function Examples", "Inlining S-Functions", and "S-Function Inlining" for further discussion of the `timestwo` example S-function.

Creating Target Folders

In this section, you 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 26-10). 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`. Note that your target and model folders should not be located anywhere in 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`. To do this 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 STF

In this section, you create an STF for your target by copying and modifying the standard STF for the ERT target. Then you validate the STF by viewing the new target in the System Target File Browser and the Configuration Parameters dialog box.

Editing the STF

To edit the STF,

1 Change your working folder to the folder you created in "Creating Target Folders" on page 26-47.

```
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 Generally, the first step in customizing an STF is to replace the header comment lines with directives that make your STF visible in the System Target File Browser and define the associated TMF (that you create shortly), make command, and External mode interface file (if any). See "Header Comments" on page 26-29 for a detailed explanation of these directives.

   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 26-40. 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 26-32.

   To set the suffix to a string 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 26-32. 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 rtwgenSettings changes made in previous steps.

   ```
   /%
     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';
   ```

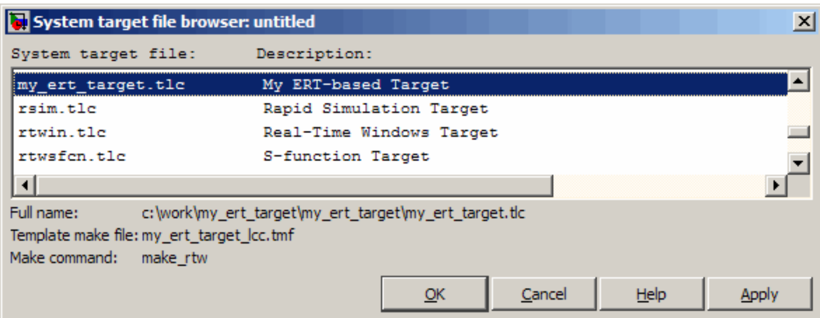
```
  rtwoptions(2).tlcvariable    = 'DummyOpt1';
  rtwoptions(2).makevariable   = '';
  rtwoptions(2).tooltip        = ['Demo option1 (non-functional)'];
  rtwoptions(2).callback       = '';

  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';

  END_RTW_OPTIONS
%/
```

10 Save your changes to `my_ert_target.tlc` and close the file.

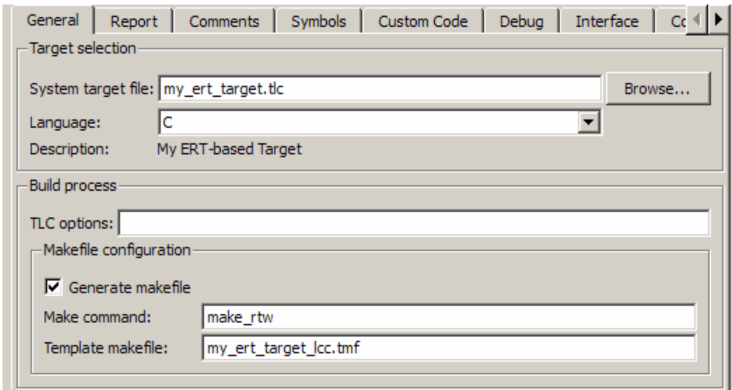
Viewing the STF

At this point, you can verify that the target inherits and displays ERT options as follows:

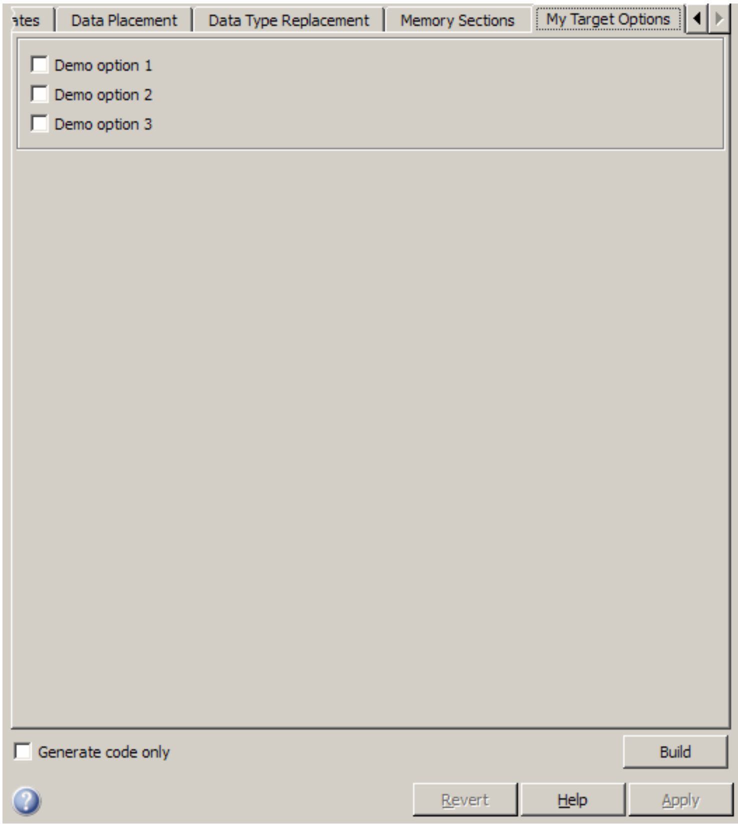
1 Create a new model.
2 Open the Model Explorer or the Configuration Parameters dialog box.
3 Select the **Code Generation** pane.
4 Click **Browse** to open the System Target File Browser.
5 In the Browser, scroll through the list of targets to find the new target, `my_ert_target.tlc`. (This step assumes that your MATLAB path contains `c:/work/my_ert_target/my_ert_target`, as previously set in “Creating Target Folders” on page 26-47.)
6 Select `My ERT-based Target` as shown below, 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**, **Make command**, and **Template makefile** fields should appear as follows:



8 Select the **My Target Options** pane and observe that the target displays the three check box options defined in the `rtwoptions` structure, as shown in the following figure.



9. Select the **Code Generation** pane and reopen the System Target File Browser.
10. Select the Embedded Coder target (`ert.tlc`) and observe that the target displays the standard ERT options.
11. Close the model. You do not need to save it.

At this point, the STF for the skeletal target is complete. Note, however, that the STF header comments reference a TMF, `my_ert_target_lcc.tmf`. You are not able to invoke the build process for your target until the TMF file is in place. In the next section, you create `my_ert_target_lcc.tmf`.

Create ERT-Based TMF

In this section, you create a TMF for your target by copying and modifying the standard ERT TMF for the LCC compiler:

1. Check that your working folder is still set to the target file folder you created previously in "Creating Target Folders" on page 26-47.

   ```
   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 to `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 of your choice.
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.

Customize Template Makefiles

In this section...
"Template Makefiles and Tokens" on page 26-56
"Invoke the make Utility" on page 26-63
"Structure of the Template Makefile" on page 26-64
"Customize and Create Template Makefiles" on page 26-67

Template Makefiles and Tokens

- "Prerequisites" on page 26-56
- "Template Makefile Role In Makefile Creation" on page 26-56
- "Template Makefile Tokens" on page 26-57

Prerequisites

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.

Template Makefile Role In Makefile Creation

TMFs are made up of statements containing tokens. The Simulink Coder 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

<table>
<tr><th>Token</th><th>Expansion</th></tr>
<tr><td colspan="2">General purpose</td></tr>
<tr><td>|>ADDITIONAL_LDFLAGS<|</td><td>Linker flags automatically added by blocks.</td></tr>
<tr><td>|>ALT_MATLAB_BIN<|</td><td>Alternate full pathname for the MATLAB executable; value is different than value for MATLAB_BIN token when the full pathname contains spaces.</td></tr>
<tr><td>|>ALT_MATLAB_ROOT<|</td><td>Alternate full pathname for the MATLAB installation; value is different than value for MATLAB_ROOT token when the full pathname contains spaces.</td></tr>
<tr><td>|>BUILDARGS<|</td><td>Options passed to make_rtw. This token is provided so that the contents of your model.mk file changes when you change the build arguments, thus forcing an update of modules when your build options change.</td></tr>
<tr><td>|>COMBINE_OUTPUT_UPDATE_FCNS<|</td><td>True (1) when Single output/update function is selected, otherwise False (0). Used for the macro definition -DONESTEPFCN=1.</td></tr>
<tr><td>|>COMPUTER<|</td><td>Computer type. See the MATLAB computer command.</td></tr>
<tr><td>|>EXPAND_LIBRARY_LOCATION<|</td><td>Location of precompiled library file. The TargetPreCompLibLocation configuration</td></tr>
</table>

<table>
<tr><th>Token</th><th>Expansion</th></tr>
<tr><td></td><td>parameter can override this setting. For examples, see “Control Library Location and Naming During Build” on page 24-7.</td></tr>
<tr><td><code>|>EXPAND_LIBRARY_NAME<|</code></td><td>Library name. For examples, see “Control Library Location and Naming During Build” on page 24-7 and “Modify the Template Makefile” on page 16-107.</td></tr>
<tr><td><code>|>EXPAND_LIBRARY_SUFFIX<|</code></td><td>Library suffix. The <code>TargetLibSuffix</code> configuration parameter can override this setting. For examples, see “Control Library Location and Naming During Build” on page 24-7.</td></tr>
<tr><td><code>|>EXT_MODE<|</code></td><td>True (1) to enable generation of External mode support code, otherwise False (0).</td></tr>
<tr><td><code>|>EXTMODE_TRANSPORT<|</code></td><td>Index of transport mechanism (for example, <code>tcpip</code>, <code>serial</code>) for External mode.</td></tr>
<tr><td><code>|>EXTMODE_STATIC<|</code></td><td>True (1) if static memory allocation is selected for External mode. False (0) if dynamic memory allocation is selected.</td></tr>
<tr><td><code>|>EXTMODE_STATIC_SIZE<|</code></td><td>Size of static memory allocation buffer (if any) for External mode.</td></tr>
<tr><td><code>|>GENERATE_ERT_S_FUNCTION<|</code></td><td>True (1) when Create SIL block is selected, otherwise False (0). Used for control of the makefile target of the build.</td></tr>
<tr><td><code>|>INCLUDE_MDL_TERMINATE_FCN<|</code></td><td>True (1) when Terminate function required is selected, otherwise False (0). Used for the macro definition <code>-DTERMFCN==1</code>.</td></tr>
<tr><td><code>|>INTEGER_CODE<|</code></td><td>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.</td></tr>
<tr><td><code>|>MAKEFILE_NAME<|</code></td><td><code>model.mk</code> — The name of the makefile that was created from the TMF.</td></tr>
<tr><td><code>|>MAT_FILE<|</code></td><td>True (1) when MAT-file logging is selected, otherwise False (0). <code>MAT_FILE</code> is a required macro</td></tr>
</table>

<table>
<tr><th>Token</th><th>Expansion</th></tr>
<tr><td></td><td>definition when compiling the source code and also is used to include logging code in the build process.</td></tr>
<tr><td><code>|>MATLAB_BIN<|</code></td><td>Location of the MATLAB executable.</td></tr>
<tr><td><code>|>MATLAB_ROOT<|</code></td><td>Path to where MATLAB is installed.</td></tr>
<tr><td><code>|>MEM_ALLOC<|</code></td><td>Either <code>RT_MALLOC</code> or <code>RT_STATIC</code>. Indicates how memory is to be allocated.</td></tr>
<tr><td><code>|>MEXEXT<|</code></td><td>MEX-file extension. See the MATLAB <code>mexext</code> command.</td></tr>
<tr><td><code>|>MODEL_MODULES<|</code></td><td>Additional generated source modules. For example, you can split a large model into two files, <code>model.c</code> and <code>model1.c</code>. In this case, this token expands to <code>model1.c</code>.</td></tr>
<tr><td><code>|>MODEL_MODULES_OBJ<|</code></td><td>Object filenames (<code>.obj</code>) corresponding to additional generated source modules.</td></tr>
<tr><td><code>|>MODEL_NAME<|</code></td><td>Name of the Simulink block diagram currently being built.</td></tr>
<tr><td><code>|>MULTITASKING<|</code></td><td>True (1) if solver mode is multitasking, otherwise False (0).</td></tr>
<tr><td><code>|>NCSTATES<|</code></td><td>Number of continuous states.</td></tr>
<tr><td><code>|>NUMST<|</code></td><td>Number of sample times in the model.</td></tr>
<tr><td><code>|>PORTABLE_WORDSIZES<|</code></td><td>True (1) when Enable portable word sizes is selected, otherwise False (0).</td></tr>
<tr><td><code>|>RELEASE_VERSION<|</code></td><td>The MATLAB release version.</td></tr>
<tr><td><code>|>S_FUNCTIONS<|</code></td><td>List of noninlined S-function sources.</td></tr>
<tr><td><code>|>S_FUNCTIONS_LIB<|</code></td><td>List of S-function libraries available for linking.</td></tr>
<tr><td><code>|>S_FUNCTIONS_OBJ<|</code></td><td>Object (<code>.obj</code>) file list corresponding to noninlined S-function sources.</td></tr>
<tr><td><code>|>SOLVER<|</code></td><td>Solver source filename, for example, <code>ode3.c</code>.</td></tr>
<tr><td><code>|>SOLVER_OBJ<|</code></td><td>Solver object (<code>.obj</code>) filename, for example, <code>ode3.obj</code>.</td></tr>
<tr><td><code>|>TARGET_LANG_EXT<|</code></td><td><code>c</code> when the Simulink Coder Language selection is C, <code>cpp</code> when the Language selection is C++. Used in the</td></tr>
</table>

<table>
<tr><th>Token</th><th>Expansion</th></tr>
<tr><td></td><td>makefile to control the extension on generated source files.</td></tr>
<tr><td><code>|>TGT_FCN_LIB<|</code></td><td>Specifies compiler command line options. The line in the makefile is <code>TGT_FCN_LIB = |>TGT_FCN_LIB<|</code>. By default, the Simulink Coder software expands the <code>|>TGT_FCN_LIB<|</code> token to indicate the default math library used in generated code. You can use this token in a makefile conditional statement to specify compiler options. Possible |>TGT_FCN_LIB<| token values are:
<table>
<tr><th>Value</th><th>Generates Calls To</th></tr>
<tr><td><code>C89/C90 (ANSI)</code></td><td>ISO®/IEC 9899:1990 C standard math library for floating-point functions</td></tr>
<tr><td><code>ISO_C</code></td><td>ISO/IEC 9899:1999 C standard math library</td></tr>
<tr><td><code>ISO_C++</code></td><td>ISO/IEC 14882:2003 C++ standard math library</td></tr>
<tr><td><code>GNU</code></td><td>GNU extensions to the ISO/IEC 9899:1999 C standard math library</td></tr>
</table></td></tr>
<tr><td><code>|>TID01EQ<|</code></td><td>True (1) if sampling rates of the continuous task and the first discrete task are equal, otherwise False (0).</td></tr>
<tr><td colspan="2">S-function and build information support

Note: For examples of the tokens in this section, see “Modify the Template Makefile” on page 16-107.</td></tr>
<tr><td><code>|>START_EXPAND_INCLUDES<|</code>
<code>|>EXPAND_DIR_NAME<|</code>
<code>|>END_EXPAND_INCLUDES<|</code></td><td>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.</td></tr>
<tr><td><code>|>START_EXPAND_LIBRARIES<|</code>
<code>|>EXPAND_LIBRARY_NAME<|</code>
<code>|>END_EXPAND_LIBRARIES<|</code></td><td>List of library names.</td></tr>
</table>

<table>
<tr><th>Token</th><th>Expansion</th></tr>
<tr><td>|>START_EXPAND_MODULES<|
|>EXPAND_MODULE_NAME<|
|>END_EXPAND_MODULES<|</td><td>Library module names within |>START_EXPAND_LIBRARIES<| and |>START_PRECOMP_LIBRARIES<| library lists.</td></tr>
<tr><td>|>START_EXPAND_RULES<|
|>EXPAND_DIR_NAME<|
|>END_EXPAND_RULES<|</td><td>Makefile rules.</td></tr>
<tr><td>|>START_PRECOMP_LIBRARIES<|
|>EXPAND_LIBRARY_NAME<|
|>END_PRECOMP_LIBRARIES<|</td><td>List of precompiled library names.</td></tr>
<tr><td colspan="2">Model reference support

Note: For examples of the tokens in this section, see “Providing Model Referencing Support in the TMF” on page 26-80.</td></tr>
<tr><td>|>MASTER_ANCHOR_DIR<|</td><td>For parallel builds, current work folder (pwd) at the time the build started.</td></tr>
<tr><td>|>MODELLIB<|</td><td>Name of the library file generated for the current model.</td></tr>
<tr><td>|>MODELREFS<|</td><td>List of models referenced by the top model.</td></tr>
<tr><td>|>MODELREF_LINK_LIBS<|</td><td>List of referenced model libraries against which the top model links.</td></tr>
<tr><td>|>MODELREF_LINK_RSPFILE_NAME<|</td><td>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 matlabroot/rtw/c/grt/grt_vc.tmf.</td></tr>
<tr><td>|>MODELREF_TARGET_TYPE<|</td><td>Type of target being built. Possible values are
• NONE: Standalone model or top model referencing other models
• RTW: Model reference Simulink Coder target build
• SIM: Model reference simulation target build</td></tr>
<tr><td>|>RELATIVE_PATH_TO_ANCHOR<|</td><td>Relative path, from the location of the generated makefile, to the MATLAB working folder.</td></tr>
</table>

<table>
<tr><th>Token</th><th>Expansion</th></tr>
<tr><td>|>START_DIR<|</td><td>Current work folder (pwd) at the time the build started. This token is required for parallel builds.</td></tr>
<tr><td>|
>START_MDLREFINC_EXPAND_INCLUDES<
|>MODELREF_INC_PATH<|
|>END_MDLREFINC_EXPAND_INCLUDES<|</td><td>List of include paths for models referenced by the top model.</td></tr>
<tr><td>|>SHARED_BIN_DIR<|</td><td>Folder for the library file built from the shared source files.</td></tr>
<tr><td>|>SHARED_LIB<|</td><td>Library file built from the shared source files, including the path to the library folder.</td></tr>
<tr><td>|>SHARED_SRC<|</td><td>Shared source files specification, including the path to the shared utilities folder.</td></tr>
<tr><td>|>SHARED_SRC_DIR<|</td><td>Folder for shared source files.</td></tr>
</table>

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 26-63
- “make Utility Versions” on page 26-63

make Command

After creating `model.mk` from your TMF, the Simulink Coder 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 26-64). You can specify additional options to `make` in the **Make command** field of the **Code Generation** pane. (See the sections “Specify a Make Command” and “Template Makefiles and Make Options” in the Simulink Coder documentation.)

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 Simulink Coder software provides the Free Software Foundation GNU `make` for both UNIX[6] and PC platforms in platform-specific subfolders under

```
matlabroot/bin
```

It is possible to use other versions of `make` with the Simulink Coder software, although GNU Make is recommended. To be compatible with the Simulink Coder software, verify that your version of `make` supports the following command format.

```
makecommand -f model.mk
```

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

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`:

```
# 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.
#
#       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 seperator 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`:

```
#------------------------ Macros read by make_rtw ------------------------------
#
# The following macros are read by the build procedure:
#
#  MAKECMD         - This is the command used to invoke the make utility
#  HOST            - What platform this template makefile is targeted for
#                    (i.e. PC or UNIX)
#  BUILD           - Invoke make from the build procedure (yes/no)?
#  SYS_TARGET_FILE - Name of system target file.

MAKECMD         = "%MATLAB%\bin\win32\gmake"
SHELL           = cmd
```

```
HOST           = PC
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.
- BUILD — Instructs make_rtw whether or not it should invoke make from the Simulink Coder 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 string 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 string found in the TMFs distributed with the bundled Simulink Coder targets (such as GRT):

  ```
    @echo ### Created executable $(MODEL).exe
  ```

  Your TMF must include either the standard build success string, 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) to use:

  - `lcc` selects `setup_for_lcc.m`
  - `vc` selects `setup_for_visual.m`
  - `vcx64` selects `setup_for_visual_x64.m`
  - `watc` selects `setup_for_watcom.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`.
- `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 string 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:

```
#--------------------- 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 26-67
- "Setting Up a Template Makefile" on page 26-68
- "Using Macros and Pattern Matching Expressions in a Template Makefile" on page 26-68
- "Using the rtwmakecfg.m API to Customize Generated Makefiles" on page 26-70
- "Supporting Continuous Time in Custom Targets" on page 26-74
- "Model Reference Considerations" on page 26-75

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 26-10 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
matlabroot/rtw/c/ert
```

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 26-27). 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 UNIX[7] environment is assumed. In a PC environment the file extensions and compile and link commands are different.

In processing the first rule

```
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 Simulink Coder 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 GNU[8] 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)
```

7. UNIX is a registered trademark of The Open Group in the United States and other countries.
8. GNU is a registered trademark of the Free Software Foundation.

```
%.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 string substitution feature for macro expansion. It replaces the source file extension (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.

Using the rtwmakecfg.m API to Customize Generated Makefiles

- "Overview" on page 26-70
- "Creating the rtwmakecfg.m Function" on page 26-71
- "Modifying the Template Makefile" on page 26-73

Overview

Simulink Coder 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 a file `rtwmakecfg.m`. The Simulink Coder software associates this file with your S-function based on its folder location. "Creating the rtwmakecfg.m Function" on page 26-71 discusses the requirements for the `rtwmakecfg` function and the data it should return.
2. Modify your target's TMF such that it supports macro expansion for the information returned by `rtwmakecfg` functions. "Modifying the Template Makefile" on page 26-73 discusses the required modifications.

After the TLC phase of the build process, when generating a makefile from the TMF, the Simulink Coder 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.

Creating the rtwmakecfg.m Function

Create the `rtwmakecfg.m` file in the same folder as your S-function component (a MEX-file with a platform-dependent extension, such as `.mexw32` on 32-bit Microsoft Windows platforms). The function must return a structured array that contains the following fields:

Field	Description
`makeInfo.includePath`	A cell array that specifies additional include folder names, organized as a row vector. The Simulink Coder build process expands the folder names into include instructions in the generated makefile.
`makeInfo.sourcePath`	A cell array that specifies additional source folder names, organized as a row vector. The Simulink Coder build process expands the folder names into make rules in the generated makefile.
`makeInfo.sources`	A cell array that specifies additional source filenames (C or C++), organized as a row vector. The Simulink Coder build process expands the filenames into make variables that contain the source files. You should specify only filenames (with extension). Specify path information with the `sourcePath` field.
`makeInfo.linkLibsObjs`	A cell array that specifies additional, fully qualified paths to object or library files against which Simulink Coder generated code should link. The Simulink Coder build process does not compile the specified objects and libraries. However, it includes them when linking the final executable. This can be useful for incorporating libraries that you do not want the Simulink Coder build process to recompile or for which the source files are not available. You might 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 Simulink Coder build process.

Field	Description
makeInfo.precompile	A Boolean flag that indicates whether the libraries specified in the rtwmakecfg.m file exist in a specified location (precompile==1) or if the libraries need to be created in the build folder during the Simulink Coder build process (precompile==0).
makeInfo.library	A structure array that specifies additional run-time libraries and module objects, organized as a row vector. The Simulink Coder build process expands the information into make rules in the generated makefile. See the next table for a list of the library fields.

The makeInfo.library field consists of the following elements:

Element	Description
makeInfo.library(n).Name	A character array that specifies the name of the library (without an extension).
makeInfo.library(n).Location	A character array that specifies the folder in which the library is located when precompiled. See the description of makeInfo.precompile in the preceding table for more information. A target can use the TargetPreCompLibLocation parameter to override this value. See “Specify the Location of Precompiled Libraries” in the Simulink Coder documentation for details.
makeInfo.library(n).Modules	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,'somdir2','lib');
makeInfo.library(1).Modules  = {'srcfile1' 'srcfile2' 'srcfile3' };
```

Note: If a path that you specify in the `rtwmakecfg.m` API contains spaces, the Simulink Coder software does not automatically convert the path to its non-space equivalent. If the build environments you intend to support do not support spaces in paths, refer to "Enable Build When Path Names Contain Spaces" in the Simulink Coder documentation.

Modifying the Template Makefile

To expand the information generated by an `rtwmakecfg` function, you can modify the following sections of your target's TMF:

- `Include Path`
- `C Flags` and/or `Additional Libraries`
- `Rules`

The TMF code examples below may not apply to your make utility. For additional examples, see the GRT or ERT TMFs located in *`matlabroot`*`/rtw/c/grt/*.tmf` or *`matlabroot`*`/rtw/c/ert/*.tmf`.

Example — Adding 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, as shown below.

```
INCLUDES = -I. -I.. $(MATLAB_INCLUDES) $(ADD_INCLUDES) $(USER_INCLUDES)
```

Example — Adding 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 on how to use configuration parameters to control library names and location during the build process, see “Control Library Location and Naming During Build” in the Simulink Coder documentation.

Example — Adding Rules to the Makefile

The following 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<|
```

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 26-78 for important information on TMF modifications you may need to make to support the Simulink Coder 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.

Support Optional Features

In this section...
"Overview" on page 26-77
"Support Model Referencing" on page 26-78
"Support Compiler Optimization Level Control" on page 26-90
"Support C Function Prototype Control" on page 26-91
"Support C++ Class Interface Control" on page 26-93
"Support Concurrent Execution of Multiple Tasks" on page 26-94

Overview

This section describes how to configure a custom embedded target to support the following optional features:

Optional Feature	Target Configuration Parameters
Building a model that includes referenced models	`ModelReferenceCompliant` `ParMdlRefBuildCompliant` (parallel build support)
Controlling the compiler optimization level for building generated code	`CompOptLevelCompliant`
Controlling the C function prototypes of initialize and step functions that are generated for a Simulink model	`ModelStepFunctionPrototypeControl-Compliant` (ERT only)
Generating and configuring C++ class interfaces to model code	`CPPClassGenCompliant` (ERT only)
Configuring concurrent execution of multiple tasks on a multicore platform	`ConcurrentExecutionCompliant`

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.

Additionally, 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.

Support Model Referencing

- “Overview” on page 26-78
- “Declaring Model Referencing Compliance” on page 26-79
- “Providing Model Referencing Support in the TMF” on page 26-80
- “Controlling Configuration Option Value Agreement” on page 26-83
- “Supporting the Shared Utilities Folder” on page 26-84
- “Verifying Worker Configuration for Parallel Builds of Model Reference Hierarchies (Optional)” on page 26-87
- “Preventing Resource Conflicts (Optional)” on page 26-89

Overview

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.

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 26-79.
- The template makefile (TMF) must define some entities that support model referencing, as described in "Providing Model Referencing Support in the TMF" on page 26-80.
- The TMF must support using the Shared Utilities folder, as described in "Supporting the Shared Utilities Folder" on page 26-84.

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" in the Simulink Coder documentation). To do this, you must modify the STF and TMF for parallel builds as described in "Declaring Model Referencing Compliance" on page 26-79 and "Providing Model Referencing Support in the TMF" on page 26-80.
- If your target supports parallel builds for large model reference hierarchies, you can additionally set up automatic verification of MATLAB Distributed Computing Server (MDCS) workers, as described in "Verifying Worker Configuration for Parallel Builds of Model Reference Hierarchies (Optional)" on page 26-87.
- 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 26-89.

See "Overview of Model Referencing" for information about model referencing in Simulink models, and "Generate Code for Referenced Models" on page 4-4 for information about model referencing in Simulink Coder generated code.

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:

```
slConfigUISetVal(hDlg,hSrc,'ModelReferenceCompliant','on');
slConfigUISetEnabled(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" in the Simulink Coder documentation.

To configure a target for parallel builds, your callback function must also set the `ParMdlRefBuildCompliant` flag as follows:

```
slConfigUISetVal(hDlg,hSrc,'ParMdlRefBuildCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'ParMdlRefBuildCompliant',false);
```

For more information about the STF callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` 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.
`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 *`matlabroot`*`/rtw/c/grt/grt_vc.tmf`.
`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 • `NONE`: Standalone model or top model referencing other models • `RTW`: Model reference Simulink Coder 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” in the Simulink Coder documentation, 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 26-26 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 26-84
- "Implementing Shared Utilities Folder Support" on page 26-85

Overview

The makefile used by the Simulink Coder 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" in the Simulink Coder documentation 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.

9. GNU is a registered trademark of the Free Software Foundation.

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 GNU[9] 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/
  - grt_lcc.tmf
  - grt_vc.tmf
  - grt_unix.tmf
- ERT: *matlabroot*/rtw/c/ert/
  - 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 UNIX[10].

```
$(SHARED_LIB) : $(SHARED_OBJS)
   @echo "### Creating $@ "
```

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

```
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 "
```

See this and other examples in the files `ert_vc.tmf`, `ert_lcc.tmf`, and `ert_unix.tmf` located at *`matlabroot`*`/rtw/c/ert`.

---

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 (MDCS) 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.

Simulink Coder 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 MDCS 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
cfg = *STF*_par_cfg_chk ('getPreferredCfg');	MDCS client	Return a structure representing the preferred configuration for MDCS workers.
[*tf*, *cfg*] = *STF*_par_cfg_chk ('getWorkerCfg', *cfg*);	MDCS workers	Each worker is passed the MDCS 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.
[*tf*, *cfg*] = *STF*_par_cfg_chk ('getCommonCfg', *cfgs*);	MDCS 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.
tf = *STF*_par_cfg_chk ('setCommonCfg', *cfg*);	MDCS 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.

Call Syntax	Called on:	Action
*STF*_par_cfg_chk ('clearCfg');	MDCS 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 MDCS client and workers.

For more information about parallel builds, see "Reduce Build Time for Referenced Models" in the Simulink Coder documentation.

Preventing Resource Conflicts (Optional)

Hook files are optional `.m` and TLC 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
```

Support Compiler Optimization Level Control

- "Overview" on page 26-90
- "Declaring Compiler Optimization Level Control Compliance" on page 26-90
- "Providing Compiler Optimization Level Control Support in the Target Makefile" on page 26-91

Overview

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 **Code Generation** pane 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 the Simulink Coder reference documentation.)

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 26-90.
- 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 26-91.

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.

Support C Function Prototype Control

- "Overview" on page 26-91
- "Declaring C Function Prototype Control Compliance" on page 26-92
- "Providing C Function Prototype Control Support in the Custom Static Main Program" on page 26-92

Overview

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 "Function Prototype Control" in the Embedded Coder documentation.)

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 26-92.
- 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 26-92.

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.

Support C++ Class Interface Control

- "Overview" on page 26-93
- "Declaring C++ Class Interface Control Compliance" on page 26-93

Overview

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 "C++ Class Interface Control" in the Embedded Coder documentation.)

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 26-93.

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:

```
slConfigUISetVal(hDlg,hSrc,'CPPClassGenCompliant','on');
slConfigUISetEnabled(hDlg,hSrc,'CPPClassGenCompliant',false);
```

For more information about the STF callback API, see the `slConfigUIGetVal`, `slConfigUISetEnabled`, and `slConfigUISetVal` 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.

Note: 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.

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.

Interface to Development Tools

In this section...

"About Interfacing to Development Tools" on page 26-96
"Makefile Approach" on page 26-97
"Interface to an Integrated Development Environment" on page 26-97

About Interfacing 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 Simulink Coder software.

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 "Makefile Approach" on page 26-97.

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 26-97.

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 Embedded Coder product provides a number of template makefiles suitable for host-based 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 26-56.

The following sections of this document supplement the basic template makefile information in the Simulink Coder documentation:

- “Supporting Multiple Development Environments” on page 26-44
- “Supplying Development Environment Information to Your Template Makefile” on page 26-24

Interface to an Integrated Development Environment

- “Introduction” on page 26-97
- “Generating a CPP_REQ_DEFINES Header File” on page 26-98
- “Interfacing to the Freescale CodeWarrior IDE” on page 26-99

Introduction

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

In Simulink Coder 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 26-56).

The Simulink Coder 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 (e.g., PurelyIntegerCode).

```
%% File: gen_rtw_req_defines_h.tlc
%openfile CPP_DEFINES = "cpp_req_defines.h"
#ifndef _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 26-99. 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 26-103. 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 26-98).

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>/Susbsystem1`.

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

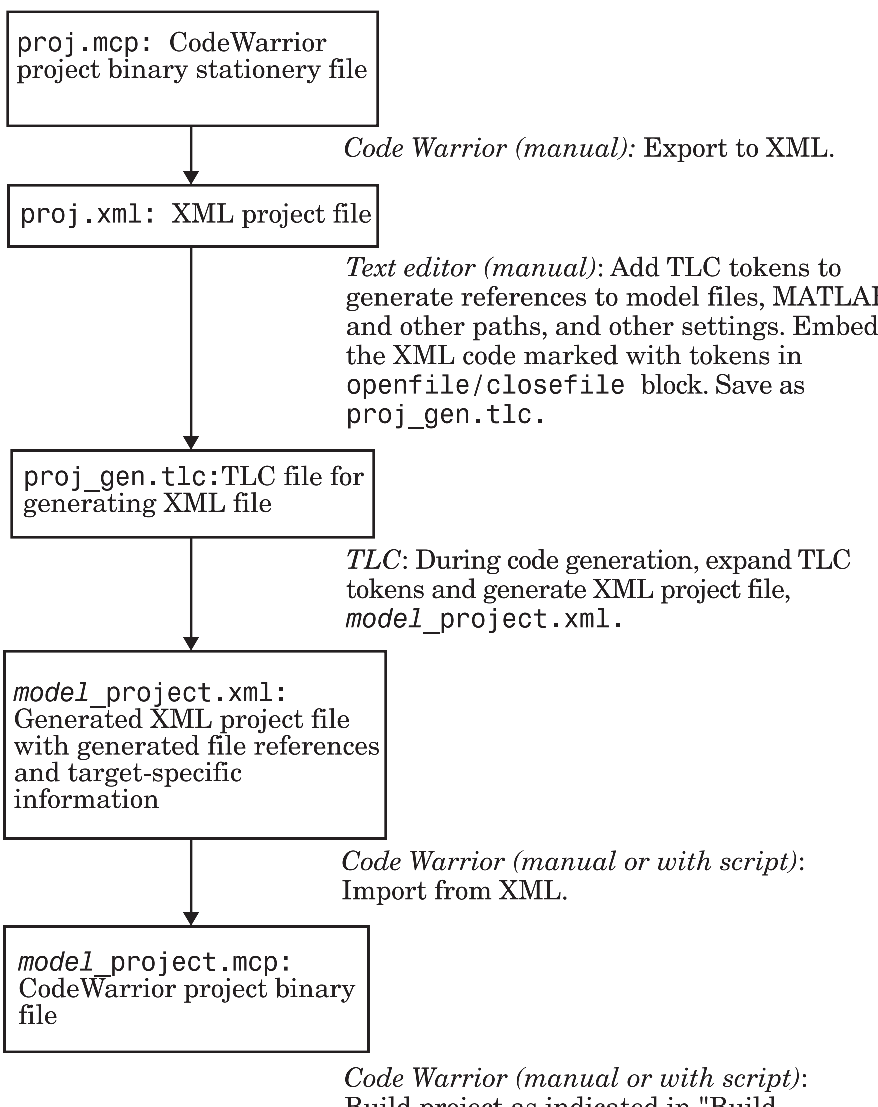
```
<file><name>%<CompiledModel.Name>.c<\name>
```

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 26-19).

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([mfilename ': Missing or empty project file argument']);
  end
  if isempty(in_qualifiedMCP)
      error([mfilename ': Missing or empty project file argument']);
  end
  ICodeWarriorApp = CreateCWComObject;
  vprint([mfilename ': 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
```

Device Drivers and Target Preferences

In this section...
"Integrate Device Drivers" on page 26-106
"Use Target Preferences" on page 26-106

Integrate 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 Simulink Coder product 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 "Integrate External Code Using Legacy Code Tool" and "Example of Integrating Existing C Functions into Simulink Models with the Legacy Code Tool".

Use Target Preferences

You may want to associate certain types of data with the target. For example, an embedded target might offer a choice of several supported development systems (cross-compilers, debuggers, and so on). To invoke a specific development tool during the build process, the target needs information such as the user's choice of development tool, and the location on the host system where the user has installed the compiler and debugger executables. Other data associated with a target might specify host/target communications parameters, such as the communications port and baud rate to be used.

Target developers need to define and store the properties they want to associate with their target. End users need a simple mechanism to set target property values.

To define preferences to associate with your target, see the following function pages:

- `addpref`
- `getpref`
- `setpref`