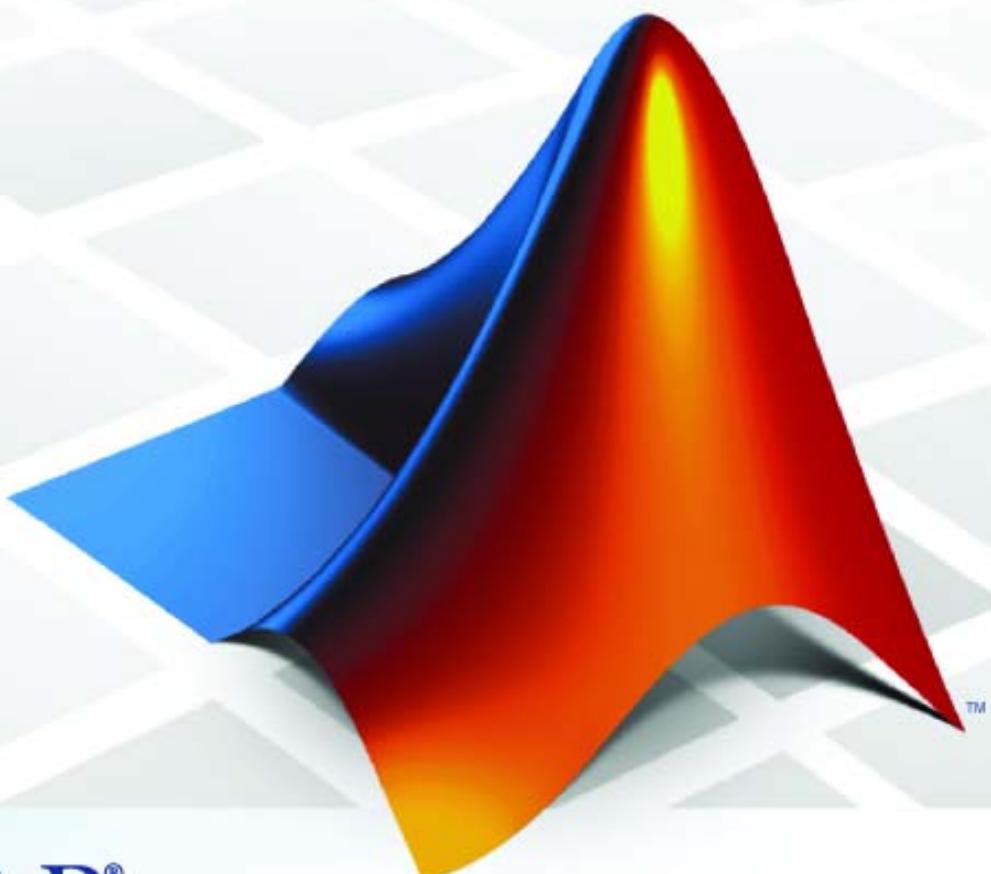


Real-Time Workshop® 7

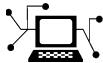
User's Guide



MATLAB®
SIMULINK®

 The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks



www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support
@	
suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

For contact information about worldwide offices, see the MathWorks Web site.

Real-Time Workshop® User's Guide

© COPYRIGHT 1994–2010 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

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

Revision History

May 1994	First printing	Version 1
January 1998	Second printing	Version 2.1
January 1999	Third printing	Version 3.11 (Release 11)
September 2000	Fourth printing	Version 4 (Release 12)
June 2001	Online only	Revised for Version 4.1 (Release 12.1)
July 2002	Online only	Revised for Version 5.0 (Release 13)
June 2004	Online only	Revised for Version 6.0 (Release 14)
October 2004	Online only	Revised for Version 6.1 (Release 14SP1)
March 2005	Online only	Revised for Version 6.2 (Release 14SP2)
September 2005	Online only	Revised for Version 6.3 (Release 14SP3)
March 2006	Online only	Revised for Version 6.4 (Release 2006a)
September 2006	Online only	Revised for Version 6.5 (Release 2006b)
March 2007	Online only	Revised for Version 6.6 (Release 2007a)
September 2007	Online only	Revised for Version 7.0 (Release 2007b)
March 2008	Online only	Revised for Version 7.1 (Release 2008a)
October 2008	Online only	Revised for Version 7.2 (Release 2008b)
March 2009	Online only	Revised for Version 7.3 (Release 2009a)
September 2009	Online only	Revised for Version 7.4 (Release 2009b)
March 2010	Online only	Revised for Version 7.5 (Release 2010a)



Contents

Introduction to Real-Time Workshop Technology

1 |

Applications of Real-Time Workshop Technology 1-2

Finding Information On How To Meet Your Goals 1-3

Developing Models for Code Generation

Setting Up Your Modeling Environment

2 |

Architecture Considerations

3 |

Laying Out the Model Architecture 3-2

Model Execution 3-2

Rapid Prototyping Program Framework 3-22

Embedded Program Framework 3-34

Creating Subsystems 3-37

Introduction 3-37

Generating Code and Executables from Subsystems 3-38

Nonvirtual Subsystem Code Generation Options 3-41

Modularity of Subsystem Code 3-50

Creating Model Components 3-51

Introduction 3-51

Overview of Referenced Model Code Generation 3-51

Project Directory Structure for Model Reference Targets	3-53
Configuring Referenced Models	3-54
Building Model Reference Targets	3-54
Real-Time Workshop Model Referencing Requirements	3-55
Storage Classes for Signals Used with Model Blocks	3-62
Inherited Sample Time for Referenced Models	3-66
Customizing the Library File Suffix, Including the File Type Extension	3-68
Real-Time Workshop Model Referencing Limitations	3-68
Creating Reusable Components	3-73
Reusable Function Option	3-73
Reusable Code and Referenced Models	3-73
Generating Reusable Code from Stateflow Charts	3-77
Generating Reusable Code for Subsystems Containing S-Function Blocks	3-77
Code Reuse Limitations	3-79
Determining Why Subsystem Code Is Not Reused	3-80
Supporting Shared Utility Directories in the Build Process	3-88
Modifying Template Makefiles to Support Shared Utilities	3-89
Creating Component Object Libraries and Enhancing Simulation Performance	3-92
Introduction	3-92
Creating an S-Function Block from a Subsystem	3-95
Tunable Parameters in Generated S-Functions	3-101
Automated S-Function Generation	3-104
System Target File and Template Makefiles	3-108
Checksums and the S-Function Target	3-109
S-Function Target Limitations	3-109
Combining Multiple Models	3-114
Using GRT Malloc to Combine Models	3-115

Block Support Considerations

4

Simulink Built-In Blocks That Support Code Generation	4-2
Block Set Support for Code Generation	4-23
User-Created Block Considerations	4-24

Scheduling Considerations

5

Introduction	5-2
Single-Tasking and Multitasking Execution Modes ...	
Introduction	5-3
Executing Multitasking Models	5-5
Multitasking and Pseudomultitasking Modes	5-6
Building a Program for Multitasking Execution	5-9
Single-Tasking Mode	5-9
Building a Program for Single-Tasking Execution	5-10
Model Execution and Rate Transitions	5-10
Simulating Models with the Simulink Product	5-11
Executing Models in Real Time	5-11
Single-Tasking Versus Multitasking Operation	5-12
Handling Rate Transitions	
Introduction	5-13
Data Transfer Problems	5-15
Data Transfer Assumptions	5-16
Rate Transition Block Options	5-16
Faster to Slower Transitions in a Simulink Model	5-21
Faster to Slower Transitions in Real Time	5-21
Slower to Faster Transitions in a Simulink Model	5-23
Slower to Faster Transitions in Real Time	5-24

Example: Single-Tasking and Multitasking Execution of a Model	5-27
Introduction	5-27
Single-Tasking Execution	5-28
Multitasking Execution	5-30
Handling Asynchronous Events	5-34
Introduction	5-34
Handling Interrupts	5-37
Rate Transitions and Asynchronous Blocks	5-52
Using Timers in Asynchronous Tasks	5-57
Creating a Customized Asynchronous Library	5-60
Asynchronous Support Limitations	5-69
Using Timers	5-70
Absolute and Elapsed Time Computation	5-70
APIs for Accessing Timers	5-72
Elapsed Timer Code Generation Example	5-77
Limitations on the Use of Absolute Time	5-80
Configuring Scheduling	5-83
Configuring Start and Stop Times	5-83
Configuring the Solver Type	5-84
Configuring the Tasking Mode	5-84

Simulation Considerations That Affect Code Generation

6

Overview	6-2
Sample Time Propagation	6-4
Constant Block Sample Times	6-5
Latches for Subsystem Blocks	6-6
Block Execution Order	6-7

Algebraic Loops	6-9
Algebraic Loops in Triggered Subsystems	6-10

Protecting Intellectual Property in Shared Model Components

7

Stateflow Considerations for Event-Driven Applications

8

Defining Data Representation and Storage for Code Generation

Deciding on Data Representation and Storage

9

Parameter Considerations

10

Introduction	10-2
Nontunable Parameter Storage	10-3
Tunable Parameter Storage	10-6
Overriding Inlined Parameters for Tuning	10-7
Tunable Parameter Storage Classes	10-8

Using the Model Parameter Configuration Dialog	
Box	10-11
Source List Panel	10-12
Global (Tunable) Parameters Panel	10-13
Declaring Tunable Variables	10-14
Tunable Expressions	10-16
Tunable Expressions in Masked Subsystems	10-16
Tunable Expression Limitations	10-18
Linear Block Parameter Tunability	10-20
Code Reuse for Subsystems with Mask Parameters	10-20
Parameter Configuration Quick Reference Diagram ..	10-22
Generated Code for Parameter Data Types	10-23
Tunable Workspace Parameter Data Type	
Considerations	10-29
Guidelines for Specifying Data Types	10-30
Limitations on Specifying Data Types in the Workspace	
Explicitly	10-30
Parameter Tuning by Using MATLAB Commands ..	10-31

Signal Considerations

11

Introduction	11-2
Signal Storage Concepts	11-3
The Global Block I/O Structure	11-3
Signals Storage Classes	11-4
Signals with Auto Storage Class	11-6
Controlling Stack Space Allocation	11-10

Signals with Test Points	11-11
Interfacing Signals to External Code	11-12
Symbolic Naming Conventions for Signals in Generated Code	11-14
Summary of Signal Storage Class Options	11-15
Monitoring Signals and Tuning Parameters	11-17
Using the C API to Monitor Signals and States and Tune Parameters	11-17
Using the Target Language Compiler API to Monitor Signals and Tune Parameters	11-17

Simulink Data Object Considerations

12

Introduction	12-2
Parameter Objects	12-4
Configuring Parameter Objects for Code Generation	12-4
Effect of Storage Classes on Code Generation for Parameter Objects	12-4
Controlling Parameter Object Code Generation with Typed Commands	12-5
Controlling Parameter Object Code Generation Using the Model Explorer	12-7
Parameter Object Configuration Quick Reference Diagram	12-11
Signal Objects	12-12
Configuring Signal Objects for Code Generation	12-12
Effect of Storage Classes on Code Generation for Signal Objects	12-12

Controlling Signal Object Code Generation By Using Typed Commands	12-13
Controlling Signal Object Code Generation By Using Model Explorer	12-15
Using Signal Objects to Initialize Signals and Discrete States	12-18
Specifying an Initial Value for a Signal Object	12-18
Signal Object Initialization in Generated Code	12-22
Tunable Initial Values	12-25
Resolving Conflicts in Configuration of Parameter and Signal Objects	12-26
Parameters	12-26
Signals and Block States	12-27
Customizing Code for Parameter and Signal Objects ..	12-29
Using Objects to Export ASAP2 Files	12-29

Enumerated Data Type Considerations

13

About Enumerated Data Types	13-2
Default Code for an Enumerated Data Type	13-3
Enumerated Type Safe Casting	13-4
Overriding Default Methods (Optional)	13-5
Specifying a Description	13-6
Specifying a Header File	13-6
Prefixing Class Names	13-7
How emlc Works with Enumerated Types	13-9

Enumerated Type Limitations	13-10
------------------------------------	-------

Block State Storage and Interfacing Considerations

14

Introduction	14-2
Block State Storage	14-2
Block State Storage Classes	14-4
Default Storage Class	14-4
Explicitly Assigned Storage Classes	14-4
Using the State Attributes Tab to Interface States to External Code	14-6
Symbolic Names for Block States	14-8
Default Block State Naming Convention	14-8
User-Defined Block State Names	14-9
Block States and Simulink Signal Objects	14-11
Summary of State Storage Class Options	14-12

Data Store Memory Considerations

15

About Simulink Data Stores	15-2
Storage Classes for Data Store Memory Blocks	15-3

Data Store Memory Blocks and Simulink Signal Objects	15-6
Nonscalar Data Stores in Generated Code	15-7
Data Store Buffering in Generated Code	15-10

Preparing Models for Code Generation

Configuring a Model for Code Generation

16

Getting Familiar With Model Configuration Options That Pertain To Code Generation	16-2
Opening the Real-Time Workshop Pane	16-2
Configuring Code Generation Objectives	16-4
Configuring a Model from the MATLAB Command Window	16-5

Enabling Instrumentation for Debugging

17

Selecting and Configuring a Target

18

About Targets and Code Formats	18-2
Targets and Code Formats	18-3

Targets and Code Styles	18-4
Backwards Compatibility of Code Formats	18-6
Types of Target Code Formats	18-10
Real-Time Code Format	18-14
Real-Time malloc Code Format	18-15
S-Function Code Format	18-17
Embedded Code Format	18-18
Available Targets	18-24
Targets Supporting Nonzero Start Time	18-29
Selecting a Target	18-29
Selecting a System Target File Programmatically	18-31
Template Makefiles and Make Options	18-32
Template Makefiles for UNIX Platforms	18-34
Template Makefiles for the Microsoft® Visual C++ Compiler	18-35
Compiler	18-35
Template Makefiles for the Watcom C/C++ Compiler	18-36
Template Makefiles for the LCC Compiler	18-38
Enabling the Real-Time Workshop Software to Build When Path Names Contain Spaces	18-39
Custom Targets	18-40

Selecting a Target Programming Language

19

Specifying Code Appearance and Documentation

20

Configuring Code Comments	20-2
Configuring Generated Identifiers	20-3
Reserved Keywords	20-4
Construction of Symbols	20-7

Customizing Targets

21

Overview of Embedded Target Development	21-2
Introducing Custom Targets	21-2
Types of Targets	21-2
Recommended Features for Embedded Targets	21-5
Example Custom Targets	21-9
Target Development Mechanics	21-11
Folder and File Naming Conventions	21-11
Components of a Custom Target	21-12
Key Directories Under the Target Root (mytarget)	21-17
Key Files in the Target Folder (mytarget/mytarget)	21-20
Additional Directories and Files for Externally Developed Targets	21-28
Understanding and Using the Build Process	21-29
Customizing System Target Files	21-37

Controlling Code Generation With the System Target	
File	21-37
System Target File Naming and Location Conventions	21-38
System Target File Structure	21-38
Defining and Displaying Custom Target Options	21-47
Tips and Techniques for Customizing Your STF	21-56
Tutorial: Creating a Custom Target Configuration	21-71
Customizing Template Makefiles	21-86
Template Makefiles and Tokens	21-86
Invoking the make Utility	21-92
Structure of the Template Makefile	21-93
Customizing and Creating Template Makefiles	21-96
Supporting Optional Features	21-109
Overview	21-109
Supporting Model Referencing	21-110
Supporting Compiler Optimization Level Control	21-122
Supporting firstTime Argument Control	21-123
Supporting C Function Prototype Control	21-126
Supporting C++ Encapsulation Interface Control	21-128
Interfacing to Development Tools	21-130
Introduction	21-130
Makefile Approach	21-131
Interfacing to an Integrated Development Environment ..	21-131
Device Drivers and Target Preferences	21-142
Integrating Device Drivers	21-142
Using Target Preferences	21-142

Generating Code and Building Executables

Generating Code

22 |

Generated Source Files and File Dependencies	22-2
Overview	22-2

Header Dependencies When Interfacing Legacy/Custom Code with Generated Code	22-4
Dependencies of the Generated Code	22-14
Specifying Include Paths in Real-Time Workshop Generated Source Files	22-19

Understanding How Code Is Generated From a Model	22-22
Model Compilation	22-22
Code Generation	22-22

Generating Reports for Code Reviews and Traceability Analysis

23

Generating a Report	23-2
Reviewing Generated Code	23-3
Viewing Generated Code in Generated HTML Reports ...	23-3
Viewing Generated Code in the Model Explorer Code Viewer	23-3

Reloading Generated Code

24

Building Executables

25

Choosing and Configuring a Compiler	25-2
Compilers and the Build Process	25-2

The Real-Time Workshop Product and ANSI ²⁸ C/C++	
Compliance	25-3
Support for C and C++ Code Generation	25-4
Support for International (Non-US-ASCII) Characters	25-5
C++ Target Language Considerations	25-8
Choosing and Configuring Your Compiler on a Microsoft Windows Platform	25-8
Choosing and Configuring Your Compiler on The Open Group UNIX Platforms	25-9
Including S-Function Source Code	25-9
Configuring the Build Process	25-10
Controlling Compiler Optimization Level and Specifying Custom Optimization Settings	25-10
Specifying TLC Options	25-11
Specifying Whether To Generate a Makefile	25-11
Specifying a Make Command	25-12
Specifying the Template Makefile	25-12
Initiating the Build Process	25-14
Files and Directories Created by the Build Process	25-15
Files Created During the Build Process	25-15
Directories Used During the Build Process	25-19
Rebuilding a Model	25-22
Reducing Build Time for Referenced Models	25-23
Understanding the Build Process	25-25
Build Process Steps	25-25
Customized Makefile Generation	25-25
Executable Program Generation	25-26
Relocating Code to Another Development Environment	25-29
Introduction	25-29
Deciding on a Structure for the Zip File	25-29

28. ANSI® is a registered trademark of the American National Standards Institute, Inc.

Deciding on a Name for the Zip File	25-30
Packaging Model Code Files in a Zip File	25-31
Inspecting the Generated Zip File	25-32
Relocating and Unpacking the Zip File	25-32
Code Packaging Example	25-32
packNGo Function Limitations	25-33

Optimizing Generated Code

26

Optimization Parameters Overview	26-2
Optimizing Models	26-5
Getting Advice About Optimizing Models for Code	
Generation	26-5
Demos Illustrating Optimizations	26-6
Minimizing Memory Requirements for Parameters and	
Data During Code Generation	26-6
Other Optimization Tools and Techniques	26-7
Minimizing Computations and Storage for Intermediate Results	26-10
Introduction	26-10
Expression Folding Example	26-11
Using and Configuring Expression Folding	26-12
Block Diagram Performance Tuning	26-15
Introduction	26-15
Lookup Tables and Polynomials	26-15
Reducing the Number of Blocks in a Model	26-28
Optimizing Code for Switch Blocks	26-32
Optimizing Data Type Usage	26-33
Additional Integer and Fixed-Point Optimizations	26-37
Optimizing Signals	26-39
Implementing Logic Signals as Boolean Data	26-39
Reducing Memory Requirements for Signals	26-39
Declaring Signals as Local Function Data	26-40
Reusing Memory Allocated for Signals	26-40

Inlining Invariant Signals	26-41
Inlining Parameters	26-43
Referenced Models	26-44
Configuring a Loop Unrolling Threshold	26-45
Optimizing Code Generated for Vector Assignments ..	26-47
Overview	26-47
Example: Using memcpy for Vector Assignments	26-48
Controlling Memory Allocation for Time Counters	26-51
Using Stack Space Allocation	26-52
Optimizing Code Resulting from Floating-Point to Integer Conversions	26-54
Removing Code That Wraps Out-of-Range Values	26-54
Removing Code That Maps NaN Values to Integer Zero ..	26-55
Improving Efficiency of Code That Uses Division by Constant Power of 2	26-56
How to Improve Efficiency of Code That Uses Division by Constant Power of 2	26-56
Rationale for Using Division or Shifts in Generated Code	26-57
When Does Real-Time Workshop Replace Division by Constant Power of 2 with Shifts?	26-57
When Does Real-Time Workshop Not Replace Division by Constant Power of 2 with Shifts?	26-58
Optimization Dependencies	26-63

Customizing Code Generation and the Build Process

27

Controlling the Compiling and Linking Phases of the Build Process	27-2
Cross-Compiling Code Generated on a Microsoft Windows System	27-4
Controlling the Location and Naming of Libraries During the Build Process	27-7
Specifying the Location of Precompiled Libraries	27-8
Controlling the Location of Model Reference Libraries	27-9
Controlling the Suffix Applied to Library File Names	27-10
Recompiling Precompiled Libraries	27-13
Customizing Post Code Generation Build Processing ..	27-14
Build Information Object	27-15
Programming a Post Code Generation Command	27-15
Defining a Post Code Generation Command	27-17
Suppressing Makefile Generation	27-18
Configuring Generated Code with TLC	27-19
Introduction	27-19
Assigning Target Language Compiler Variables	27-19
Setting Target Language Compiler Options	27-21
Customizing the Target Build Process with the STF_make_rtw Hook File	27-22
Overview	27-22
File and Function Naming Conventions	27-22
STF_make_rtw_hook.m Function Prototype and Arguments	27-23
Applications for STF_make_rtw_hook.m	27-27
Using STF_make_rtw_hook.m for Your Build Procedure ..	27-28
Customizing the Target Build Process with sl_customization.m	27-29

Overview	27-29
Registering Build Process Hook Functions Using sl_customization.m	27-31
Variables Available for sl_customization.m Hook Functions	27-32
Example Build Process Customization Using sl_customization.m	27-32
Replacing the STF_rtw_info_hook Mechanism	27-34

Converting MATLAB Code to C/C++ Code

28

About Embedded MATLAB Coder	28-3
Converts MATLAB Code to C/C++ Code	28-3
Running a Demo of Embedded MATLAB Coder	28-4
How Embedded MATLAB Coder Resolves Function Calls	28-4
Workflows for Converting MATLAB Code to C/C++ Code	28-5
Workflow for Converting MATLAB Code to Embeddable C/C++ Code	28-5
Workflow for Converting MATLAB Code to a MEX Function	28-6
Installing Prerequisite Products for Embedded MATLAB Coder	28-8
Setting Up the C/C++ Compiler	28-9
How to Set Up Your C/C++ Compiler	28-9
Supported Compilers for Generating MEX Functions	28-9
Supported Compilers for Generating C/C++ Code	28-9
File Paths and Naming Conventions	28-11
Compile Path Search Order	28-11
Can I Add Files to the Embedded MATLAB Path?	28-11
When to Use the Embedded MATLAB Path	28-11

Adding Folders to Search Paths	28-12
Naming Conventions	28-12
Making MATLAB Code Compliant with the Embedded MATLAB Subset	28-14
Debugging Strategies	28-14
Detecting Embedded MATLAB Syntax Violations at Compile Time	28-15
Configuring Your Environment for Code Generation	28-16
Types of Configuration Objects	28-16
Working with Configuration Objects	28-18
Creating Configuration Objects	28-19
Modifying Configuration Objects	28-20
Saving Configuration Objects	28-21
Specifying Properties of Primary Function Inputs	28-23
Why You Must Specify Input Properties	28-23
Properties to Specify	28-23
Rules for Specifying Properties of Primary Inputs	28-27
Methods for Defining Properties of Primary Inputs	28-27
Defining Input Properties by Example at the Command Line	28-28
Defining Input Properties Programmatically in the MATLAB File	28-31
Choosing Your Target	28-41
Types of Targets	28-41
Specifying the Target to the Compiler	28-42
Specifying a Language for Embeddable Code Generation	28-43
Relationship of Targets and Configuration Objects	28-43
Location of Generated Files	28-44
Specifying main Functions for C/C++ Executables	28-44
Compiling Your MATLAB File	28-46
Before Compiling Your MATLAB Code	28-46
Running Embedded MATLAB Coder	28-46
Specifying Compiler Options	28-47
Specifying Custom Files to Build	28-47

Compiling More Than One Entry-Point MATLAB Function	28-48
Advantages of Compiling More Than One Entry-Point Function	28-48
Function	28-48
How to Compile More Than One Entry-Point Function	28-48
How to Call an Entry-Point Function in a MEX Function	28-50
How to Call an Entry-Point Function in a C/C++ Library	
Function from C/C++ Code	28-50
Generating C/C++ Code from MATLAB Code That Uses Global Data	28-52
Workflow Overview	28-52
Declaring Global Variables	28-52
Defining Global Data	28-53
Synchronizing Global Data with MATLAB	28-54
Limitations of Using Global Data	28-58
How emlc Generates Code	28-59
Partitioning Generated Files	28-59
How to Select the File Partitioning Method	28-59
Partitioning Generated Files When You Generate One	
C/C++ File Per MATLAB File	28-60
Generated Files and Locations	28-65
File Partitioning and Inlining	28-67
Working with Compilation Reports	28-70
About Compilation Reports	28-70
Generating Compilation Reports	28-70
Names and Locations of Compilation Reports	28-71
Opening Compilation Reports	28-72
Description of Compilation Reports	28-73
Viewing Your MATLAB Code	28-74
Viewing Call Stack Information	28-75
Viewing the Generated C/C++ Code	28-77
Viewing the Compilation Summary Information	28-77
Viewing Error and Warning Messages	28-77
Viewing Variables in Your MATLAB Code	28-79
Viewing Target Build Information	28-85
Keyboard Shortcuts for the Compilation Report	28-85
Embedded MATLAB Compilation Report Limitations	28-86
Calling Generated C/C++ Functions	28-88

Conventions for Calling Functions in Generated Code	28-88
Calling C/C++ Functions from the Embedded MATLAB	
Subset	28-88
Calling Initialize and Terminate Functions	28-89
Calling C/C++ Functions with Multiple Outputs	28-94
Calling C/C++ Functions that Return Arrays	28-94
Integrating Custom C/C++ Code with Generated Code	28-95
About Custom C/C++ Code Integration with Embedded MATLAB Coder	28-95
Specifying Custom C/C++ Files on the Command Line	28-95
Specifying Custom C/C++ Files with Configuration Objects	28-95
Speeding Up Simulation with the Basic Linear Algebra Subprograms (BLAS) Library	28-98
How Embedded MATLAB Functions Use the BLAS Library	28-98
When to Disable BLAS Library Support	28-98
How to Disable BLAS Library Support	28-99
Supported Compilers	28-99
Controlling Runtime Checks	28-101
Types of Runtime Checks	28-101
When to Disable Runtime Checks	28-102
How to Disable Runtime Checks	28-102
Debugging Runtime Errors	28-104
Viewing Errors in the Runtime Stack	28-104
Handling Runtime Errors	28-105

Integrating External Code With Generated C and C++ Code

About External Code Integration

29

Integration Options	29-2
Introduction	29-2
Types of External Code Integration	29-2
Reusing Algorithmic Components in Generated Code	29-5
Examples of Reusable Algorithmic Components	29-5
Integrating External MATLAB Code	29-6
Integrating External C or C++ Code	29-9
Integrating Fortran Code	29-13
Other Integration Considerations for Reusable Algorithmic Components	29-13
Deploying Algorithm Code Within a Target Environment	29-16
Generating Target Optimizations Within Algorithm Code	29-21
Exporting Generated Algorithm Code for Embedded Applications	29-23
Exporting Algorithm Executables for System Simulation	29-26
Making External Code Language Compatible With Generated Code	29-27

Integrating External Code Using Model Configuration Parameters

30

Integrating External Code Using Custom Code Blocks

31

Custom Code Library	31-2
Example: Using a Custom Code Block	31-6
Custom Code in Subsystems	31-9
Preventing User Source Code from Being Deleted from Build Directories	31-10

Integrating External Code Using S-Functions

32

Introduction to S-Functions for Code Generation	32-2
About S-Functions	32-2
Additional Information	32-3
Classes of Problems Solved by S-Functions	32-3
Types of S-Functions	32-4
Basic Files Required for Implementation	32-7
Guidelines for Writing S-Functions for Use with Real-Time Workshop Software	32-8
Writing Noninlined S-Functions	32-9
About Noninlined S-Functions	32-9
Guidelines for Writing Noninlined S-Functions	32-9
Noninlined S-Function Parameter Type Limitations	32-10

Writing Wrapper S-Functions	32-12
About Wrapper S-Functions	32-12
MEX S-Function Wrapper	32-12
TLC S-Function Wrapper	32-17
The Inlined Code	32-22
Writing Fully Inlined S-Functions	32-23
Multiport S-Function Example	32-23
Automating Generation of Files for Fully Inlined S-Functions Using Legacy Code Tool	32-25
Legacy Code Tool and Code Generation	32-25
Generating Inlined S-Function Files for Code Generation Support	32-26
Applying Model Code Style Settings to Legacy Functions ..	32-27
Addressing Dependencies on Files in Different Locations	32-28
Deploying Generated S-Functions for Simulation and Code Generation	32-29
Writing Fully Inlined S-Functions with the mdlRTW Routine	32-30
About S-Functions and mdlRTW	32-30
S-Function RTWdata	32-31
The Direct-Index Lookup Table Algorithm	32-32
The Direct-Index Lookup Table Example	32-33
Guidelines for Writing Inlined S-Functions	32-56
Writing S-Functions That Support Expression Folding	32-57
Introduction	32-57
Categories of Output Expressions	32-58
Acceptance or Denial of Requests for Input Expressions ..	32-63
Utilizing Expression Folding in Your TLC Block Implementation	32-67
Writing S-Functions That Specify Port Scope and Reusability	32-73

Writing S-Functions That Specify Sample Time	
Inheritance Rules	32-79
Writing S-Functions That Support Code Reuse	32-81
Writing S-Functions for Multirate Multitasking	
Environments	32-82
Introduction	32-82
Rate Grouping Support in S-Functions	32-82
Creating Multitasking-Safe, Multirate, Port-Based Sample Time S-Functions	32-83
Build Support for S-Functions	32-90
Introduction	32-90
Implicit Build Support	32-91
Specifying Additional Source Files for an S-Function	32-91
Using TLC Library Functions	32-93
Using the <code>rwtmakecfg.m</code> API to Customize Generated Makefiles	32-93

Setting Up Generated Code To Interface With Components in the Run-Time Environment

Configuring the Target Environment

33

Configuring the Hardware Implementation	33-2
Describing the Emulation and Embedded Targets	33-3
Describing Embedded Hardware Characteristics	33-12
Describing Emulation Hardware Characteristics	33-13
Configuring the Target Hardware Environment	33-16
Specifying Target Interfaces	33-16
Selecting and Viewing Target Function Libraries	33-20

Interfacing With a Real-Time Operating System

34 |

Resources for Wind River Systems Tornado

Applications	34-2
Introduction	34-2
The Tornado Environment	34-2
The Tornado Target	34-3
Block Library for Wind River Systems VxWorks	34-3

Wind River Systems Tornado Application

Architecture	34-4
Hardware Architecture	34-4
Software Architecture	34-5
Module Architecture	34-6
Host Processes	34-6
Wind River Systems VxWorks Tasks	34-7

Installing the Wind River Systems Tornado

Software	34-10
Introduction	34-10
Installing and Configuring the Tornado Environment	34-10
Connecting to the Wind River Systems VxWorks Target ..	34-10
Verifying the Tornado Installation	34-11

Implementing a Wind River Systems Tornado

Application	34-12
Designing the Simulink Model	34-12
Adding Device Driver Blocks	34-12
Specifying Code Generation Options	34-13
Configuring the Template Makefile	34-19

Building the Application

34-22

Automatic Download and Execution

34-22

Manual Download and Execution

34-24

Introduction	34-24
Resetting Wind River Systems VxWorks System	34-24
Downloading Files	34-24

Starting the Program	34-25
----------------------------	-------

Interacting with Target Application Data Using the C API

35

Introduction	35-2
Generating C API Files	35-3
Selecting C API with the Configuration Parameters Dialog Box	35-3
Selecting C API from the MATLAB Command Line	35-4
Generating C API and ASAP2 Files	35-4
Description of C API Files	35-5
Overview	35-5
Structure Arrays Generated in C API Files	35-8
Generating Example C API Files	35-9
C API Signals	35-11
C API States	35-15
C API Parameters	35-16
Mapping C API Data Structures to the Real-Time Model Data Structure	35-19
Using the C API in an Application	35-21
Example: Using the C API to Access Model Signals and States	35-21
Example: Using the C API to Access Model Parameters ..	35-28
C API Limitations	35-33

Generating Model Information for Host-Based ASAP2 Data Measurement and Calibration

36

Overview	36-2
Targets Supporting ASAP2	36-3
Defining ASAP2 Information	36-4
Memory Address Attribute	36-5
Generating an ASAP2 File	36-7
Introduction	36-7
Using Generic Real-Time Target or Embedded Coder	
Target	36-7
Using the ASAM-ASAP2 Data Definition Target	36-9
Generating ASAP2 Files for Referenced Models	36-11
Customizing an ASAP2 File	36-12
Introduction	36-12
ASAP2 File Structure on the MATLAB Path	36-12
Customizing the Contents of the ASAP2 File	36-13
ASAP2 Templates	36-14
Structure of the ASAP2 File	36-19
Generating ASAP2 and C API Files	36-21

Communicating With Code Executing on a Target System Using Simulink External Mode

37

Introduction	37-3
Using the External Mode User Interface	37-5
External Mode Interface Options	37-5

External Mode Related Menu and Toolbar Items	37-8
External Mode Control Panel	37-13
Target Interfacing	37-15
External Signal Uploading and Triggering	37-17
Data Archiving	37-22
Parameter Downloading	37-24
External Mode Compatible Blocks and Subsystems	37-27
Compatible Blocks	37-27
Signal Viewing Subsystems	37-28
Supported Blocks for Data Archiving	37-29
External Mode Communications Overview	37-31
Introduction	37-31
Download Mechanism	37-31
Inlined and Tunable Parameters	37-33
Client/Server Implementations	37-35
Introduction	37-35
Using the TCP/IP Implementation	37-35
Using the Serial Implementation	37-38
Running the External Program	37-41
Implementing an External Mode Protocol Layer	37-43
Using External Mode Programmatically	37-44
Creating a TCP/IP Transport Layer for External Communication	37-50
Introduction	37-50
Design of External Mode	37-50
External Mode Communications Overview	37-54
External Mode Source Files	37-55
Implementing a Custom Transport Layer	37-59
External Mode Limitations	37-67
Limitations on Changing Parameters	37-67
Limitation on Mixing 32-bit and 64-bit Architectures	37-68
Limitations on Uploading Data	37-68
Limitations on Archiving Data	37-68

Accessing Memory in Generated Code

38

Setting Up Runtime Logging to MAT-Files

39

About Logging to MAT-Files	39-2
Configuring a Model to Log States, Time, and Output	39-3
Overriding the Default MAT-File Name	39-3
Overriding the Default MAT-File Variable Names	39-4
Logging Data with Scope and To Workspace Blocks ..	39-5
Logging Data with To File Blocks	39-6
Data Logging Differences in Single- and Multitasking Models	39-7

Sharing Utility Functions

40

About Shared Utility Functions	40-2
Controlling Shared Utility Function Generation	40-3
rtwtypes.h and Shared Utility Functions	40-4
Incremental Shared Utility Function Generation and Compilation	40-4

Shared Utility Checksum	40-6
Shared Fixed-Point Utility Functions	40-8

Verifying Generated Code Applications

Testing and Refining Concept Models With Host-Based Simulations

41

About Host-Based Simulation	41-2
Host-Based Simulation Performance	41-3
Goals of Host-Based Simulation	41-4
Running Host-Based Simulations	41-5

Testing and Refining Concept Models With Standalone Rapid Simulations

42

About Rapid Simulation	42-2
Rapid Simulation Performance	42-3
General Rapid Simulation Workflow	42-4
Identifying Your Rapid Simulation Requirements	42-6
Configuring Import Blocks to Provide Rapid Simulation Source Data	42-8

Configuring and Building a Model for Rapid Simulation	42-9
Setting Up Rapid Simulation Input Data	42-12
Introduction	42-12
Creating a MAT-File That Includes a Model Parameter Structure	42-13
Creating a MAT-File for a From File Block	42-17
Creating a MAT-File for an Inport Block	42-18
Programming Scripts for Batch and Monte Carlo Simulations	42-23
Running Rapid Simulations	42-24
Introduction	42-24
Requirements for Running Rapid Simulations	42-26
Setting a Clock Time Limit for a Rapid Simulation	42-26
Overriding a Model Simulation Stop Time	42-27
Reading the Parameter Vector into a Rapid Simulation ..	42-28
Specifying New Signal Data File for a From File Block ..	42-28
Specifying Signal Data File for an Inport Block	42-31
Changing Block Parameters for an RSim Simulation	42-34
Specifying a New Output File Name for a Simulation	42-36
Specifying New Output File Names for To File Blocks	42-36
Rapid Simulation Target Limitations	42-37

Verifying Generated Code

43 |

About Generated Code Verification	43-2
Tracing Generated Code	43-3
About Code Traceability	43-3
Format of Traceability Tags	43-3
Examples of Tagged Code	43-4
Tracing Code To Blocks Using hilite_system	43-4
Traceability Limitations	43-5

Profiling Code Performance	43-7
About Profiling Code Performance	43-7
How to Profile Code Performance	43-7
Profiling Limitation	43-9

Testing and Refining a Model With Rapid Prototyping

44

About Rapid Prototyping	44-2
Rapid Prototyping System Simulations	44-4
About System Simulation	44-4
Goals of System Simulation	44-4
Running System Simulations	44-5
Rapid Prototyping a Real-Time System	44-7
About Real-Time Rapid Prototyping	44-7
Goals of Real-Time Rapid Prototyping	44-8
Refining Component Code With Real-Time Rapid Prototyping	44-8

Verifying Generated Source Code With Software-In-the-Loop Simulation

45

About Software-In-the-Loop Simulation	45-2
Maintaining Bit-True Agreement Between Host Simulations and Code for Target Deployment	45-3
Setting Up and Running SIL Simulations Using a Hand-Written S-Function Wrapper	45-5

Verifying a System With Hardware-In-the-Loop Simulation

46

About Hardware-In-the-Loop Simulation **46-2**

Setting Up and Running HIL Simulations **46-4**

Troubleshooting

A

Troubleshooting Compiler Configurations	A-2
Compiler Version Mismatch Errors	A-2
Generated Executable Image Produces Incorrect Results	A-2
Compile-Time Errors	A-3

Examples

B

Model Reference	B-2
Models	B-2
Timing Services	B-2
Data Management	B-2
Model Code Packaging	B-3
Optimizations	B-3

Advanced Code Generation	B-3
Custom Code	B-3
S-Functions	B-3
Interfaces	B-4

Index

Introduction to Real-Time Workshop Technology

- “Applications of Real-Time Workshop Technology” on page 1-2
- “Finding Information On How To Meet Your Goals” on page 1-3

Applications of Real-Time Workshop Technology

You have a variety of ways for applying Real-Time Workshop technology with the Real-Time Workshop® and Real-Time Workshop® Embedded Coder™ products. For an introduction, in the Real-Time Workshop getting started information, see the following topics:

- “What You Can Accomplish Using Real-Time Workshop Technology”
- “How the Technology Can Fit Into Your Development Process”
- “How You Can Apply the Technology to the V-Model for System Development”

Finding Information On How To Meet Your Goals

The following tables list goals that you might have, as you apply Real-Time Workshop technology, and where to find guidance on how to meet those goals. Each table focuses on goals that pertain to a step of the V-model for system development.

- Documenting and Validating Requirements on page 1-3
- Developing a Model Executable Specification on page 1-6
- Developing a Detailed Software Design on page 1-9
- Generating the Application Code on page 1-15
- Integrating and Verifying Software on page 1-20
- Integrating, Verifying, and Calibrating System Components on page 1-25

Documenting and Validating Requirements

Goals	Related Product Information	Demos
Capture requirements in a document, spreadsheet, data base, or requirements management tool	Simulink® Report Generator™ documentation Third-party vendor tools such as Microsoft® Word, Microsoft® Excel®, raw HTML, or IBM® Rational® DOORS®	
Associate requirements documents with objects in concept models Generate a report on requirements associated with a model	Simulink® Verification and Validation™ “Requirements Linking and Traceability” in the Simulink Verification and Validation documentation Bidirectional tracing in Microsoft Word, Microsoft Excel, HTML, and Telelogic® DOORS	slvnvdemo_fuelsys_docreq

Documenting and Validating Requirements (Continued)

Goals	Related Product Information	Demos
Include requirements tags in generated code	Simulink Verification and Validation “Including Requirements Information with Generated Code” in the Simulink Verification and Validation documentation	<code>rtwdemo_requirements</code>
Trace model blocks and subsystems to generated code and vice versa	Real-Time Workshop Embedded Coder “Tracing Generated Code” on page 43-3 “Verifying Generated Code” in the Real-Time Workshop Embedded Coder documentation	<code>rtwdemo_hyperlinks</code>
Verify, refine, and test concept model in non-real time on a host system	Developing Models for Code Generation on page 1 in the Real-Time Workshop documentation “Developing Models for Code Generation” in the Real-Time Workshop Embedded Coder documentation “Running Simulations” and “Accelerating Models” in the Simulink® documentation	<code>rtwdemo_f14</code> <code>rtwdemo_fuelsys_publish</code>

Documenting and Validating Requirements (Continued)

Goals	Related Product Information	Demos
<p>Run standalone rapid simulations</p> <p>Run batch or Monte-Carlo simulations</p> <p>Repeat simulations with varying data sets, interactively or programmatically with scripts, without rebuilding the model</p> <p>Tune parameters and monitor signals interactively</p> <p>Simulate models for hybrid dynamic systems that include components and an environment or plant that requires variable-step solvers and zero-crossing detection</p>	<p>Chapter 42, “Testing and Refining Concept Models With Standalone Rapid Simulations”</p> <p>Chapter 37, “Communicating With Code Executing on a Target System Using Simulink External Mode”</p>	<pre>rtwdemo_rsim_param_survey_ script rtwdemo_rsim_batch_script rtwdemo_rsim_param_tuning</pre>
Distribute simulation runs across multiple computers	SystemTest™ SystemTest documentation MATLAB® Distributed Computing Server™ documentation Parallel Computing Toolbox™ documentation	

Developing a Model Executable Specification

Goals	Related Product Information	Demos
Produce design artifacts for algorithms that you develop in MATLAB® code for reviews and archiving	MATLAB® Report Generator™ “MATLAB Report Generator” documentation	
Produce design artifacts from Simulink and Stateflow® models for reviews and archiving	Simulink Report Generator “Simulink Report Generator” System Design Description Report	<code>rtwdemo_codegenrpt</code>
Add one or more components to another environment for system simulation Refine a component model Refine an integrated system model Verify functionality of a model in non-real time Test a concept model	Chapter 44, “Testing and Refining a Model With Rapid Prototyping”	
Schedule generated code	Chapter 5, “Scheduling Considerations” “Handling Asynchronous Events” on page 5-34	<code>rtwdemos</code> , select Multirate Support folder
Specify function boundaries of systems	“Creating Subsystems” on page 3-37	<code>rtwdemo_atomic</code> <code>rtwdemo_ssreuse</code> <code>rtwdemo_filepart</code> <code>rtwdemo_export_functions</code>
Specify components and boundaries for design and incremental code generation	Chapter 3, “Architecture Considerations”	<code>rtwdemo_mdlreftop</code>

Developing a Model Executable Specification (Continued)

Goals	Related Product Information	Demos
Specify function interfaces so that external software can compile, build, and invoke the generated code	<p>Real-Time Workshop Embedded Coder</p> <p>“Controlling Generation of Function Prototypes” and “Controlling Generation of Encapsulated C++ Model Interfaces” in the Real-Time Workshop Embedded Coder documentation</p>	<p><code>rtdemo_fcnprotoctrl</code> <code>rtdemo_cppencap</code></p>
Manage data packaging in generated code for integrating and packaging data	<p>Real-Time Workshop Embedded Coder</p> <p>“Defining Data Representation and Storage for Code Generation” in the Real-Time Workshop Embedded Coder documentation</p> <p>“Relocating Code to Another Development Environment” on page 25-29 (PacknGo)</p>	<p><code>rtdemos</code>, select Function, File and Data Packaging folder</p>
Generate and control the format of comments and identifiers in generated code	<p>Real-Time Workshop Embedded Coder</p> <p>“Customizing Comments in Generated Code” and “Configuring Symbols” in the Real-Time Workshop Embedded Coder documentation</p>	<p><code>rtdemo_comments</code> <code>rtdemo_symbols</code></p>

Developing a Model Executable Specification (Continued)

Goals	Related Product Information	Demos
Create a zip file that contains generated code files, static files, and dependent data to build generated code in an environment other than your host computer	“Relocating Code to Another Development Environment” on page 25-29 (PacknGo)	<code>rtwdemo_buildinfo</code>
Export models for validation in a system simulator using shared libraries	Real-Time Workshop Embedded Coder “Creating and Using Host-Based Shared Libraries” in the Real-Time Workshop Embedded Coder documentation	<code>rtwdemo_shrlib</code>
Refine component and environment model designs by rapidly iterating between algorithm design and prototyping Verify whether a component can adequately control a physical system in non-real time Evaluate system performance before laying out hardware, coding production software, or committing to a fixed design Test hardware	“Rapid Prototyping a Real-Time System” on page 44-7 Developing Models for Code Generation on page 1 in the Real-Time Workshop documentation “Developing Models for Code Generation” in the Real-Time Workshop Embedded Coder documentation “Profiling Code Performance” on page 43-7	<code>rtwdemo_profile</code>

Developing a Model Executable Specification (Continued)

Goals	Related Product Information	Demos
Generate code for rapid prototyping	Chapter 18, "Selecting and Configuring a Target" Real-Time Workshop Embedded Coder "Developing Models for Code Generation" in the Real-Time Workshop Embedded Coder documentation Chapter 34, "Interfacing With a Real-Time Operating System"	<code>rtdemo_counter</code> <code>rtdemo_async</code>
Generate code for rapid prototyping in hard real time, using PCs	xPC Target™ xPC Target documentation	<code>doc xpcdemos</code>
Generate code for rapid prototyping in soft real time, using PCs	Real-Time Windows Target™ Real-Time Windows Target documentation	<code>rtvdp</code> (and others)

Developing a Detailed Software Design

Goals	Related Product Information	Demos
Refine a model design for representation and storage of data in generated code	Defining Data Representation and Storage for Code Generation on page 1 "Relocating Code to Another Development Environment" on page 25-29 (PacknGo)	

Developing a Detailed Software Design (Continued)

Goals	Related Product Information	Demos
Select a deployment code format	Chapter 18, “Selecting and Configuring a Target” Real-Time Workshop Embedded Coder “Developing Models for Code Generation” and “Generating Code for AUTOSAR Software Components” in the Real-Time Workshop Embedded Coder documentation Chapter 34, “Interfacing With a Real-Time Operating System”	<code>rtwdemo_counter</code> <code>rtwdemo_async</code> <code>rtwdemo_osek</code> “AUTOSAR Demos” in the Real-Time Workshop Embedded Coder documentation
Specify target hardware settings	Chapter 18, “Selecting and Configuring a Target”	<code>rtwdemo_targetsettings</code>
Design model variants	“Using Model Reference Variants” in the Simulink documentation	
Specify fixed-point algorithms in Simulink, Stateflow, and Embedded MATLAB®	Simulink® Fixed Point™ “Data Types and Scaling” and “Code Generation” in the Simulink Fixed Point documentation	<code>rtwdemo_fixpt1</code> <code>rtwdemo_fuelsys_fxp_publish</code>
Convert a floating-point model or subsystem to a fixed-point representation	Simulink Fixed Point “Fixed-Point Advisor” in the Simulink Fixed Point documentation	<code>fxpdemo_fpa</code>
Iterate to obtain an optimal fixed-point design, using autoscaling	Simulink Fixed Point “Automatic Scaling” in the Simulink Fixed Point documentation	<code>fxpdemo_feedback</code>

Developing a Detailed Software Design (Continued)

Goals	Related Product Information	Demos
Create or rename data types specifically for your application	Real-Time Workshop Embedded Coder “Code Generation with User-Defined Data Types” in the Real-Time Workshop Embedded Coder documentation	rtdemo_udt
Control the format of identifiers in generated code	Real-Time Workshop Embedded Coder “Configuring Symbols” in the Real-Time Workshop Embedded Coder documentation	rtdemo_symbols
Specify how signals, tunable parameters, block states, and data objects are declared, stored, and represented in generated code	Real-Time Workshop Embedded Coder “Creating and Using Custom Storage Classes” in the Real-Time Workshop Embedded Coder documentation	rtdemo_cscpredef
Create a data dictionary for a model	Real-Time Workshop Embedded Coder “Managing Data Definitions and Declarations With the Data Dictionary” in the Real-Time Workshop Embedded Coder documentation	rtdemo_advsc
Relocate data segments for generated functions and data using #pragmas for calibration or safe data access	Real-Time Workshop Embedded Coder “Inserting Comments and Pragmas in Generated Code” in the Real-Time Workshop Embedded Coder documentation	rtdemo_memsec

Developing a Detailed Software Design (Continued)

Goals	Related Product Information	Demos
Assess and adjust model configuration parameters based on the application and an expected run-time environment	Preparing Models for Code Generation on page 1 in the Real-Time Workshop documentation “Preparing Models for Code Generation” in the Real-Time Workshop Embedded Coder documentation	rtdemo_f14
Check a model against basic modeling guidelines	“Consulting the Model Advisor” in the Simulink documentation	rtdemo_advisor1
Add custom checks to the Simulink Model Advisor	Simulink Verification and Validation “Customizing the Model Advisor” in the Simulink Verification and Validation documentation	slvndemo_mdladv
Check a model against custom standards or guidelines	“Consulting the Model Advisor” in the Simulink documentation	
Check a model against industry standards and guidelines (MathWorks™ Automotive Advisory Board (MAAB), IEC 61508, and DO-178B)	Real-Time Workshop Embedded Coder “Developing Models and Code That Comply with Industry Standards and Guidelines” in the Real-Time Workshop Embedded Coder documentation Simulink Verification and Validation “Model Advisor Checks” in the Simulink Verification and Validation documentation	rtdemo_iec61508

Developing a Detailed Software Design (Continued)

Goals	Related Product Information	Demos
Obtain model coverage for structural coverage analysis such as MC/DC	Simulink Verification and Validation “Using Model Coverage” in the Simulink Verification and Validation documentation	<code>cvbasic_operation</code>
Prove properties and generate test vectors for models	Simulink® Design Verifier™ Simulink Design Verifier documentation	<code>sldvdemo_cruise_control</code> <code>sldvdemo_cruise_control_verification</code>
Generate reports of models and software designs	MATLAB Report Generator MATLAB Report Generator documentation Simulink Report Generator Simulink Report Generator System Design Description Report documentation	<code>rtwdemo_codegenrpt</code>
Conduct reviews of your model and software designs with coworkers, customers, and suppliers who do not have Simulink available	Simulink Report Generator “Exporting Simulink Models to Web Views” and “Comparing XML Files Exported from Simulink Models” in the Simulink Report Generator documentation	<code>slxml_sfcar</code>

Developing a Detailed Software Design (Continued)

Goals	Related Product Information	Demos
<p>Refine the concept model of your component or system</p> <p>Test and validate the model functionality in real time</p> <p>Test the hardware</p> <p>Obtain real-time profiles and code metrics for analysis and sizing based on your embedded processor</p> <p>Assess the feasibility of the algorithm based on integration with the environment or plant hardware</p>	<p>“Rapid Prototyping On a Target System” in the Real-Time Workshop Embedded Coder documentation</p> <p>Embedded IDE Link™</p> <p>“Embedded IDE Link” documentation</p> <p>Target Support Package™</p> <p>“Target Support Package” documentation</p>	
Generate source code for your models, integrate the code into your production build environment, and run it on existing hardware	<p>“About System Simulation” on page 44-4</p> <p>Chapter 18, “Selecting and Configuring a Target”</p> <p>Real-Time Workshop Embedded Coder</p> <p>“Developing Models for Code Generation” and “Generating Code for AUTOSAR Software Components” in the Real-Time Workshop Embedded Coder documentation</p> <p>Chapter 34, “Interfacing With a Real-Time Operating System”</p>	<p><code>rtwdemo_counter</code></p> <p><code>rtwdemo_fcnprotoctrl</code></p> <p><code>rtwdemo_cppencap</code></p> <p><code>rtwdemo_async</code></p> <p><code>rtwdemo_osek</code></p> <p>“AUTOSAR Demos” in the Real-Time Workshop Embedded Coder documentation</p>

Developing a Detailed Software Design (Continued)

Goals	Related Product Information	Demos
Integrate existing externally written C or C++ code with your model for simulation and code generation	<p>“Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation</p> <p>Integrating External Code With Generated C and C++ Code on page 1 in the Real-Time Workshop documentation</p> <p>“Integrating External Code and Generated C and C++ Code” in the Real-Time Workshop Embedded Coder documentation</p>	rtdemos, select Integrating with C Code or Integrating with C++ Code folder
Generate code for on-target rapid prototyping on specific embedded microprocessors and IDEs	<p>Embedded IDE Link</p> <p>“Embedded IDE Link” documentation</p> <p>Target Support Package</p> <p>“Target Support Package” documentation</p>	See help for link and target support package products

Generating the Application Code

Goals	Related Product Information	Demos
Optimize generated ANSI® C code for production (for example, disable floating-point code, remove termination and error handling code, and combine code entry points into single functions)	<p>Chapter 26, “Optimizing Generated Code”</p> <p>Real-Time Workshop Embedded Coder</p> <p>“Preparing Models for Code Generation” in the Real-Time Workshop Embedded Coder documentation</p>	rtdemos, select Optimizations folder

Generating the Application Code (Continued)

Goals	Related Product Information	Demos
Optimize code for a specific run-time environment, using specialized function libraries	Real-Time Workshop Embedded Coder “Replacing Math Functions and Operators Using Target Function Libraries” in the Real-Time Workshop Embedded Coder documentation	<code>rtwdemo_tfl_script</code>
Control the format and style of generated code	Real-Time Workshop Embedded Coder “Controlling Code Style” in the Real-Time Workshop Embedded Coder documentation	<code>rtwdemo_parentheses</code>
Control the comments inserted into generated code	Real-Time Workshop Embedded Coder “Customizing Comments in Generated Code” in the Real-Time Workshop Embedded Coder	<code>rtwdemo_comments</code>
Enter special instructions or tags for postprocessing by third-party tools or processes	“Customizing Post Code Generation Build Processing” on page 27-14	<code>rtwdemo_buildinfo</code>
Include requirements tags in generated code	Simulink Verification and Validation “Including Requirements Information with Generated Code” in the Simulink Verification and Validation documentation	<code>rtwdemo_requirements</code>

Generating the Application Code (Continued)

Goals	Related Product Information	Demos
Trace model blocks and subsystems to generated code and vice versa	<p>Real-Time Workshop Embedded Coder</p> <p>“Generating Reports for Code Reviews and Traceability Analysis”, “Tracing Code to Model Objects Using Hyperlinks”, “Tracing Model Objects to Generated Code”, and “Developing Models and Code That Comply with Industry Standards and Guidelines” in the Real-Time Workshop Embedded Coder documentation</p>	<p><code>rtdemo_comments</code> <code>rtdemo_hyperlinks</code></p>
Integrate existing externally written code with code generated for a model	<p>“Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation</p> <p>Integrating External Code With Generated C and C++ Code on page 1 in the Real-Time Workshop documentation</p> <p>“Integrating External Code and Generated C and C++ Code” in the Real-Time Workshop Embedded Coder documentation</p>	<p><code>rtdemos</code>, select Integrating with C Code or Integrating with C++ Code folder</p>

Generating the Application Code (Continued)

Goals	Related Product Information	Demos
Verify generated code for MISRA C® ¹ and other run-time violations	Real-Time Workshop Embedded Coder “Developing Models and Code That Comply with MISRA C Guidelines” in the Real-Time Workshop Embedded Coder documentation Documentation for PolySpace® Products	rtdemo_polySpace
Protect the intellectual property of component model design and generated code Generate a binary file (shared library)	Chapter 7, “Protecting Intellectual Property in Shared Model Components”	
Generate a MEX-file S-function for a model or subsystem so that it can be shared with a third-party vendor	“Creating Component Object Libraries and Enhancing Simulation Performance” on page 3-92	
Generate a shared library for a model or subsystem so that it can be shared with a third-party vendor	“Creating and Using Host-Based Shared Libraries” in the Real-Time Workshop Embedded Coder documentation	

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

Generating the Application Code (Continued)

Goals	Related Product Information	Demos
Test generated production code with an environment or plant model to verify a successful conversion of the model to code	<p>Chapter 45, “Verifying Generated Source Code With Software-In-the-Loop Simulation” in the Real-Time Workshop documentation</p> <p>“Verifying Generated Source Code With Software-in-the-Loop Simulation” in the Real-Time Workshop Embedded Coder documentation</p>	<code>rtdemo_sil_pil_script</code>
Write or generate an S-function wrapper for calling your generated source code from a model running in Simulink	<p>“Writing Wrapper S-Functions” on page 32-12</p> <p>Real-Time Workshop Embedded Coder</p> <p>“Generating S-Function Wrappers” in the Real-Time Workshop Embedded Coder documentation</p>	<code>rtdemo_sil_pil_script</code>
Set up and run SIL tests on your host computer	<p>Chapter 45, “Verifying Generated Source Code With Software-In-the-Loop Simulation” in the Real-Time Workshop documentation</p> <p>Real-Time Workshop Embedded Coder</p> <p>“Verifying Generated Source Code With Software-in-the-Loop Simulation” in the Real-Time Workshop Embedded Coder documentation</p>	<code>rtdemo_sil_pil_script</code>

Integrating and Verifying Software

Goals	Related Product Information	Demos
Integrate existing externally written C or C++ code with a model for simulation and code generation	<p>“Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation</p> <p>Integrating External Code With Generated C and C++ Code on page 1 in the Real-Time Workshop documentation</p> <p>“Integrating External Code and Generated C and C++ Code” in the Real-Time Workshop Embedded Coder documentation</p>	<code>rtdemos</code> , select Integrating with C Code or Integrating with C++ Code folder
Connect to data interfaces for generated C code data structures	<p>Setting Up Generated Code To Interface With Components in the Run-Time Environment on page 1 in the Real-Time Workshop documentation</p> <p>“Setting Up Generated Code To Interface With Components in the Run-Time Environment” in the Real-Time Workshop Embedded Coder documentation</p>	<code>rtdemo_capi</code> <code>rtdemo_asap2</code>

Integrating and Verifying Software (Continued)

Goals	Related Product Information	Demos
Control the generation of code interfaces so that external software can compile, build, and invoke the generated code	Real-Time Workshop Embedded Coder “Controlling Generation of Function Prototypes” and “Controlling Generation of Encapsulated C++ Model Interfaces” in the Real-Time Workshop Embedded Coder documentation	<code>rtwdemo_fcnprotoctrl</code> <code>rtwdemo_cppencap</code>
Export virtual and function-call subsystems	Real-Time Workshop Embedded Coder “Exporting Function-Call Subsystems” in the Real-Time Workshop Embedded Coder documentation	<code>rtwdemo_export_functions</code>
Include target-specific code	Real-Time Workshop Embedded Coder “Replacing Math Functions and Operators Using Target Function Libraries” in the Real-Time Workshop Embedded Coder documentation	<code>rtwdemo_tfl_script</code>
Customize and control the build process	Chapter 25, “Building Executables”	<code>rtwdemo_buildinfo</code>
Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer	“Relocating Code to Another Development Environment” on page 25-29 (PacknGo)	<code>rtwdemo_buildinfo</code>

Integrating and Verifying Software (Continued)

Goals	Related Product Information	Demos
Integrate all software components as a complete system for testing in the target environment	“Verifying a Component by Building a Complete Real-Time Target Environment” in the Real-Time Workshop Embedded Coder documentation	
Generate source code for integration with specific production environments	Chapter 18, “Selecting and Configuring a Target” Real-Time Workshop Embedded Coder “Developing Models for Code Generation” and “Generating Code for AUTOSAR Software Components” in the Real-Time Workshop Embedded Coder documentation Chapter 34, “Interfacing With a Real-Time Operating System”	<code>rtwdemo_async</code> <code>rtwdemo_osek</code> “AUTOSAR Demos” in the Real-Time Workshop Embedded Coder documentation
Integrate code for a specific run-time environment, using specialized function libraries	Real-Time Workshop Embedded Coder “Replacing Math Functions and Operators Using Target Function Libraries” in the Real-Time Workshop Embedded Coder documentation	<code>rtwdemo_tfl_script</code>
Enter special instructions or tags for postprocessing by third-party tools or processes	“Customizing Post Code Generation Build Processing” on page 27-14	<code>rtwdemo_buildinfo</code>

Integrating and Verifying Software (Continued)

Goals	Related Product Information	Demos
Integrate existing externally written code with code generated for a model	<p>“Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation</p> <p>Integrating External Code With Generated C and C++ Code on page 1 in the Real-Time Workshop documentation</p> <p>“Integrating External Code and Generated C and C++ Code” in the Real-Time Workshop Embedded Coder documentation</p>	rtwdemos, select Integrating with C Code or Integrating with C++ Code folder
Connect to data interfaces for the generated C code data structures	<p>Setting Up Generated Code To Interface With Components in the Run-Time Environment on page 1 in the Real-Time Workshop documentation</p> <p>“Setting Up Generated Code To Interface With Components in the Run-Time Environment” in the Real-Time Workshop Embedded Coder documentation</p>	rtwdemo_capi rtwdemo_asap2
Customize and control the build process	Chapter 25, “Building Executables”	rtwdemo_buildinfo

Integrating and Verifying Software (Continued)

Goals	Related Product Information	Demos
Create a zip file that contains generated code files, static files, and dependent data needed to build the generated code in an environment other than your host computer	“Relocating Code to Another Development Environment” on page 25-29 (PacknGo)	<code>rtwdemo_buildinfo</code>
Schedule the generated code	Chapter 5, “Scheduling Considerations”	<code>rtwdemos</code> , select Multirate Support folder
Verify object code files in a target environment	“Verifying Compiled Object Code with Processor-in-the-Loop Simulation” in the Real-Time Workshop Embedded Coder documentation	<code>rtwdemo_sil_pil_script</code>
Set up and run PIL tests on your target system	“Verifying Compiled Object Code with Processor-in-the-Loop Simulation” in the Real-Time Workshop Embedded Coder documentation “Embedded IDE Link” documentation	<code>rtwdemo_sil_pil_script</code> <code>rtwdemo_custom_pil</code> <code>rtwdemo_rtiostream</code> See the list of supported hardware for the Real-Time Workshop Embedded Coder product on the MathWorks Web site, and then find a demo for the related product of interest

Integrating, Verifying, and Calibrating System Components

Goals	Related Product Information	Demos
<p>Integrate the software and its microprocessor with the hardware environment for the final embedded system product</p> <p>Add the complexity of the environment (or plant) under control to the test platform</p> <p>Test and verify the embedded system or control unit by using a real-time target environment</p>	<p>Chapter 46, “Verifying a System With Hardware-In-the-Loop Simulation”</p>	
Generate source code for HIL testing	<p>Chapter 18, “Selecting and Configuring a Target ”</p> <p>Real-Time Workshop Embedded Coder</p> <p>“Developing Models for Code Generation” and “Generating Code for AUTOSAR Software Components” in the Real-Time Workshop Embedded Coder documentation</p> <p>Chapter 34, “Interfacing With a Real-Time Operating System ”</p>	rtwdemo_f14
Conduct hard real-time HIL testing using PCs	xPC Target xPC Target documentation	doc xpcdemos

Integrating, Verifying, and Calibrating System Components (Continued)

Goals	Related Product Information	Demos
Ensure ECU is properly tuned for its intended use	Setting Up Generated Code To Interface With Components in the Run-Time Environment on page 1 in the Real-Time Workshop documentation “Setting Up Generated Code To Interface With Components in the Run-Time Environment” in the Real-Time Workshop Embedded Coder documentation	
Generate ASAP2 data files	Chapter 36, “Generating Model Information for Host-Based ASAP2 Data Measurement and Calibration”	rtdemo_asap2
Generate C API data interface files	Chapter 35, “Interacting with Target Application Data Using the C API”	rtdemo_capi

Developing Models for Code Generation

- Chapter 2, “Setting Up Your Modeling Environment”
- Chapter 3, “Architecture Considerations”
- Chapter 4, “Block Support Considerations”
- Chapter 5, “Scheduling Considerations”
- Chapter 6, “Simulation Considerations That Affect Code Generation”
- Chapter 7, “Protecting Intellectual Property in Shared Model Components”
- Chapter 8, “Stateflow Considerations for Event-Driven Applications”



Setting Up Your Modeling Environment

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

The following table guides you to information and demos that pertain to use of Real-Time Workshop technology to meet goals for specific domains.

Goals	Related Product Information	Demos
Specify algorithms as MATLAB code	Embedded MATLAB documentation	<code>rtwdemo_emlcbasicdemo</code>
Specify algorithms graphically as Simulink models for controls design	“Creating a Model” in the Simulink documentation	<code>rtwdemo_f14</code>
Call Embedded MATLAB functions in Simulink	“MATLAB Function Blocks” in the Simulink documentation	<code>rtwdemo_eml_aero_radar</code>
Model event-driven designs, such as finite state machines and truth tables, in Simulink for fault detection, modes, and conditional logic	Stateflow “Creating Stateflow Charts” in the Stateflow documentation	<code>rtwdemo_fuelsys</code>
Model signal processing filters (for example, fast Fourier transform (FFT) and infinite impulse response (IIR)) in Simulink	Signal Processing Blockset™ Signal Processing Blockset documentation	<code>rtwdemo_lmsadeq</code>
Model video processing models in Simulink	Video and Image Processing Blockset™ Video and Image Processing Blockset documentation	
Create physical models or plant models in Simulink	Simscape™ Simscape documentation	
Model other domains and applications	All products supported by code generation	

Architecture Considerations

- “Laying Out the Model Architecture” on page 3-2
- “Creating Subsystems” on page 3-37
- “Creating Model Components” on page 3-51
- “Creating Reusable Components” on page 3-73
- “Supporting Shared Utility Directories in the Build Process” on page 3-88
- “Creating Component Object Libraries and Enhancing Simulation Performance” on page 3-92
- “Combining Multiple Models” on page 3-114

Laying Out the Model Architecture

In this section...

“Model Execution” on page 3-2

“Rapid Prototyping Program Framework” on page 3-22

“Embedded Program Framework” on page 3-34

Model Execution

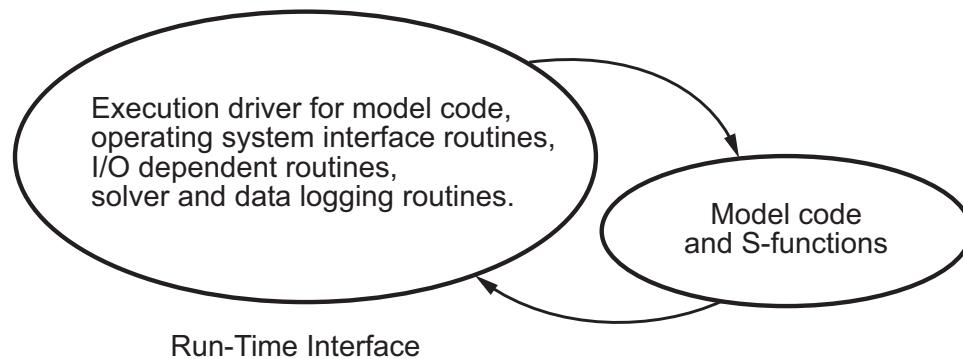
- “Introduction” on page 3-2
- “Models for Non-Real-Time Single-Tasking Systems” on page 3-4
- “Models for Non-Real-Time Multitasking Systems” on page 3-5
- “Models for Real-Time Single-Tasking Systems” on page 3-6
- “Models for Real-Time Multitasking Systems” on page 3-8
- “Models for Multitasking Systems that Use Real-Time Tasking Primitives” on page 3-10
- “Program Timing” on page 3-11
- “Program Execution” on page 3-13
- “External Mode Communication” on page 3-13
- “Data Logging in Single-Tasking and Multitasking Model Execution” on page 3-13
- “Rapid Prototyping and Embedded Model Execution Differences” on page 3-14
- “Rapid Prototyping Model Functions” on page 3-15
- “Embedded Model Functions” on page 3-21

Introduction

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 Real-Time Workshop 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 Real-Time Workshop 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. Function names used in ERT and GRT targets are shown, followed by the comparable GRT-compatible calls in parentheses.

- **Initialization:** *model_initialize* (*MdlInitializeSizes*, *MdlInitializeSampleTimes*, *MdlStart*) initializes the run-time interface code and the model code.
- **ModelOutputs:** Calling all blocks in your model that have a sample hit at the current time and having them produce their output. *model_output* (*MdlOutputs*) 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* (*MdlUpdate*) calls all 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:** Calling all blocks in your model that have continuous states and having them update their derivatives. *model_derivatives* is only called in minor time steps.
- **ModelTerminate:** *model_terminate* (*MdlTerminate*) 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.

- “Models for Non-Real-Time Single-Tasking Systems” on page 3-4
- “Models for Non-Real-Time Multitasking Systems” on page 3-5
- “Models for Real-Time Single-Tasking Systems” on page 3-6
- “Models for Real-Time Multitasking Systems” on page 3-8
- “Models for Multitasking Systems that Use Real-Time Tasking Primitives” on page 3-10

For a complete set of correspondences between GRT and ERT function identifiers, see the table Identifiers for Real-Time Model Data Structure Variants on page 3-30.

Models for 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 any 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.

Models for 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 there were no preemption in a real-time system.

Multitasking execution assumes that all tasks 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*.

Models for 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 these steps must occur before the next interrupt.

Models for 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.
        ModelDerivatives
        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.

Models for Multitasking Systems that Use 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.
            ModelDerivatives
            Do 0 or more
```

```

        ModelOutputs(tid=0)
        ModelDerivatives
        EndDo (number of iterations depends upon the solver).
        Integrate derivatives to update continuous states.
    EndLoop
EndMainLoop
}
main()
{
    Initialization
    Start/spawn task "tSubRate".
    Start/spawn task "tBaseRate".

    Start clock that does a "semGive" on a clockSem semaphore.
    Wait on "model-running" semaphore.
    Shutdown
}

```

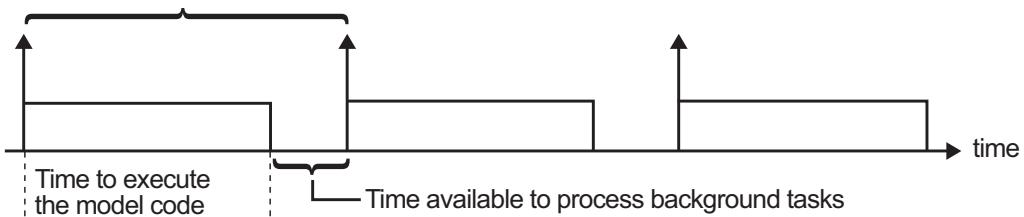
In this multitasking environment, the model is executed using real-time operating system tasking primitives. Such environments require several model tasks (`tBaseRate` and several `tSubRate` tasks) to run the model code. The base rate task (`tBaseRate`) has a higher priority than the substrate tasks. The substrate task for `tid=1` has a higher priority than the substrate 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

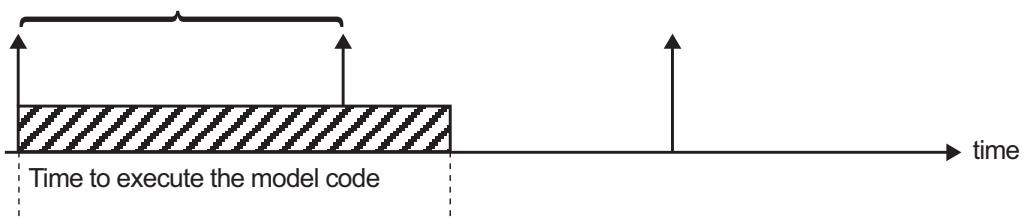
Real-time programs require careful timing of the task invocations (either by using an interrupt or a real-time operating system tasking primitive) to ensure that the model code executes to completion before another task invocation occurs. This includes time to read and write data to and from external hardware.

The next figure illustrates interrupt timing.

Sample interval is appropriate for this model code execution.



Sample interval is too short for this model code execution.



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 the `while` loop never exits), then the shutdown code never executes.

For more information on how the timing engine works, see “Using Timers” on page 5-70.

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 at the appropriate time to ensure real-time execution of the model code.

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 Real-Time Workshop data-logging features, described in Chapter 17, "Enabling Instrumentation for Debugging", 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, all blocks that have a hit at time t execute, whereas in multitasking, `LogTXY` is called after `ModelOutputs(tid=0)`, which executes only the blocks that have a hit at time t and that have a task identifier of 0. This results in differences in the logged values between single-tasking and multitasking logging. Specifically, consider a model with two sample times, the faster sample time having a period of 1.0 second and the slower sample time having a period of 10.0

seconds. At time $t = k*10$, $k=0,1,2\dots$ both the fast ($tid=0$) and slow ($tid=1$) blocks execute. When executing in multitasking mode, when `LogTXY` is called, the slow blocks execute, but the previous value is logged, whereas in single-tasking the current value is logged.

Another difference occurs when logging data in an enabled subsystem. Consider an enabled subsystem that has a slow signal driving the enable port and fast blocks within the enabled subsystem. In this case, the evaluation of the enable signal occurs in a slow task, and the fast blocks see a delay of one sample period; thus the logged values will show these differences.

To summarize differences in logged data between single-tasking and multitasking, differences will be seen when

- Any root 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 Real-Time Workshop 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 a detailed discussion of how generated embedded code executes, see the Real-Time Workshop Embedded Coder documentation.

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) needed 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 at appropriate times. 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. The solver is called to updates
- **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 the appropriate section of `MdlOutput`.

$$x_{d+1} = f_u(t, x_d, u)$$

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 \dot{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 Real-Time Workshop software generates.

The next example shows the general contents of the rapid prototyping style of C code written to the `model.c` file.

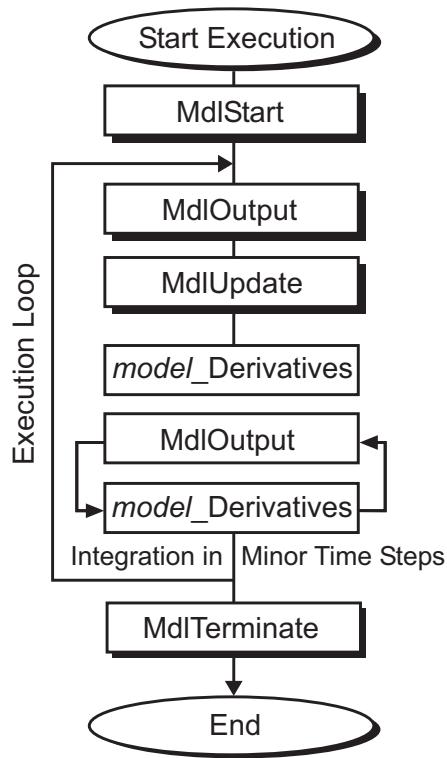
```
/*
 * Version, Model options, TLC options,
 * and code generation information are placed here.
 */
<includes>
void MdlStart(void)
{
    /*
     * State initialization code.
     * Model start-up code - one time initialization code.
     * Execute any block enable methods.
     * Initialize output 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. */

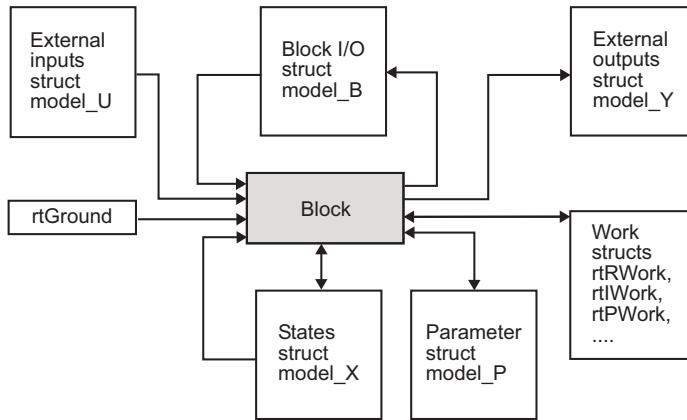
    /* Compute: dxc = fd(t,xc,u) for each block in model_derivatives
       as needed. */
}
void MdlTerminate(void)
{
    /* Perform shutdown code for any blocks that
       have a termination action */
}
```

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 Real-Time Workshop products reduce the size of the *model_B* structure by
 - Reusing the entries in the *model_B* structure
 - Making other entries local variables

See Chapter 11, “Signal Considerations” 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 any 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 all 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 Import block signals. Field names are determined by either the block's output signal name, when present, or by the Import 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.

Embedded Model Functions

The Real-Time Workshop Embedded Coder target generates the following functions:

- *model_initialize*: Performs all model initialization and should be called once before you start executing your model.
- If the **Single output/update function** code generation option is selected, then you see
 - *model_step*: Contains the output and update code for all blocks in your model.
 Otherwise, you see
 - *model_output*: Contains the output code for all blocks in your model.
 - *model_update*: Contains the update code for all blocks in your model.
- If the **Terminate function required** code generation option is selected, then you see
 - *model_terminate*: This contains all model shutdown code and should be called as part of system shutdown.

See “Model Entry Points” in the Real-Time Workshop Embedded Coder documentation for complete descriptions of these functions.

Rapid Prototyping Program Framework

- “Introduction” on page 3-22
- “Rapid Prototyping Program Architecture” on page 3-22
- “Rapid Prototyping System-Dependent Components” on page 3-23
- “Rapid Prototyping System-Independent Components” on page 3-25
- “Rapid Prototyping Application Components” on page 3-28

Introduction

The code modules generated from a Simulink model—*model.c* (or *.cpp*), *model.h*, and other files — implement the model’s system equations, contain block parameters, and perform initialization.

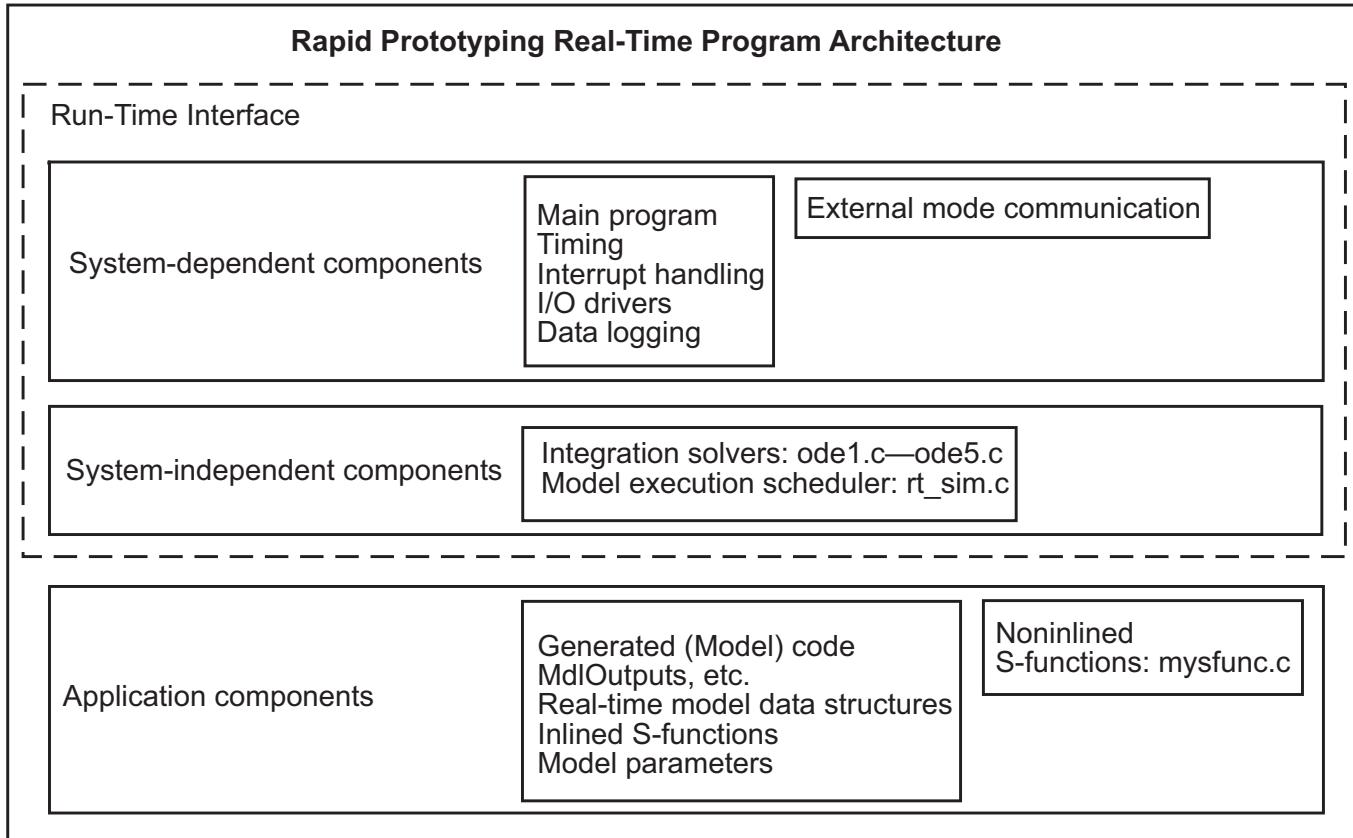
The Real-Time Workshop program framework provides the additional source code necessary to build the model code into a complete, standalone program. The program framework consists of *application modules* (files containing source code to implement required functions) designed for a number of different programming environments.

The automatic program builder ensures that the program is created with the proper modules once you have configured your template makefile. The application modules and the code generated for a Simulink model are implemented using a common API. This API defines a data structure (called a *real-time model*, sometimes abbreviated as *rtM*) that encapsulates all data for your model.

This API is similar to that of S-functions, with one major exception: the API assumes that there is only one instance of the model, whereas S-functions can have multiple instances. The function prototypes also differ from S-functions.

Rapid Prototyping Program Architecture

The structure of a real-time program consists of three components. Each component has a dependency on a different part of the environment in which the program executes. The next figure shows this structure.



The Real-Time Workshop architecture consists of three parts, the first two of which include system-dependent components and system-independent components. Together these two parts form the *run-time interface*.

This architecture adapts readily to a wide variety of environments by isolating the dependencies of each program component. The following sections discuss each component in more detail and include descriptions of the application modules that implement the functions carried out by the system-dependent components, system-independent components, and application components.

Rapid Prototyping System-Dependent Components

These components contain the program's main function, which controls program timing, creates tasks, installs interrupt handlers, enables data logging, and performs error checking.

The way in which application modules implement these operations depends on the type of computer. This means that, for example, the components used for a PC-based program perform the same operations, but differ in method of implementation from components designed to run on a VME target.

The main Function. The `main` function in a C/C++ program is the point where execution begins. In Real-Time Workshop application programs, the `main` function must perform certain operations. These operations can be grouped into three categories: initialization, model execution, and program termination.

Initialization.

- Initialize special numeric parameters `rtInf`, `rtMinusInf`, and `rtNaN`. These are variables that the model code can use.
- Call the model registration function to get a pointer to the real-time model. The model registration function has the same name as your model. It is responsible for initializing real-time model fields and any S-functions in your model.
- Initialize the model size information in the real-time model. This is done by calling `MdlInitializeSizes`.
- Initialize a vector of sample times and offsets (for systems with multiple sample rates). This is done by calling `MdlInitializeSampleTimes`.
- Get the model ready for execution by calling `MdlStart`, which initializes states and similar items.
- Set up the timer to control execution of the model.
- Define background tasks and enable data logging, if selected.

Model Execution.

- Execute a background task: for example, communicate with the host during external mode simulation or introduce a wait state until the next sample interval.
- Execute model (initiated by interrupt).
- Log data to buffer (if data logging is used).

- Return from interrupt.

Program Termination.

- Call a function to terminate the program if it is designed to run for a finite time — destroy the real-time model data structure, deallocate memory, and write data to a file.

Rapid Prototyping Application Modules for System-Dependent Components.

Components. The application modules contained in the system-dependent components generally include a main module such as `rt_main.c`, containing the main entry point for C. There can also be additional application modules for such things as I/O support and timer handling.

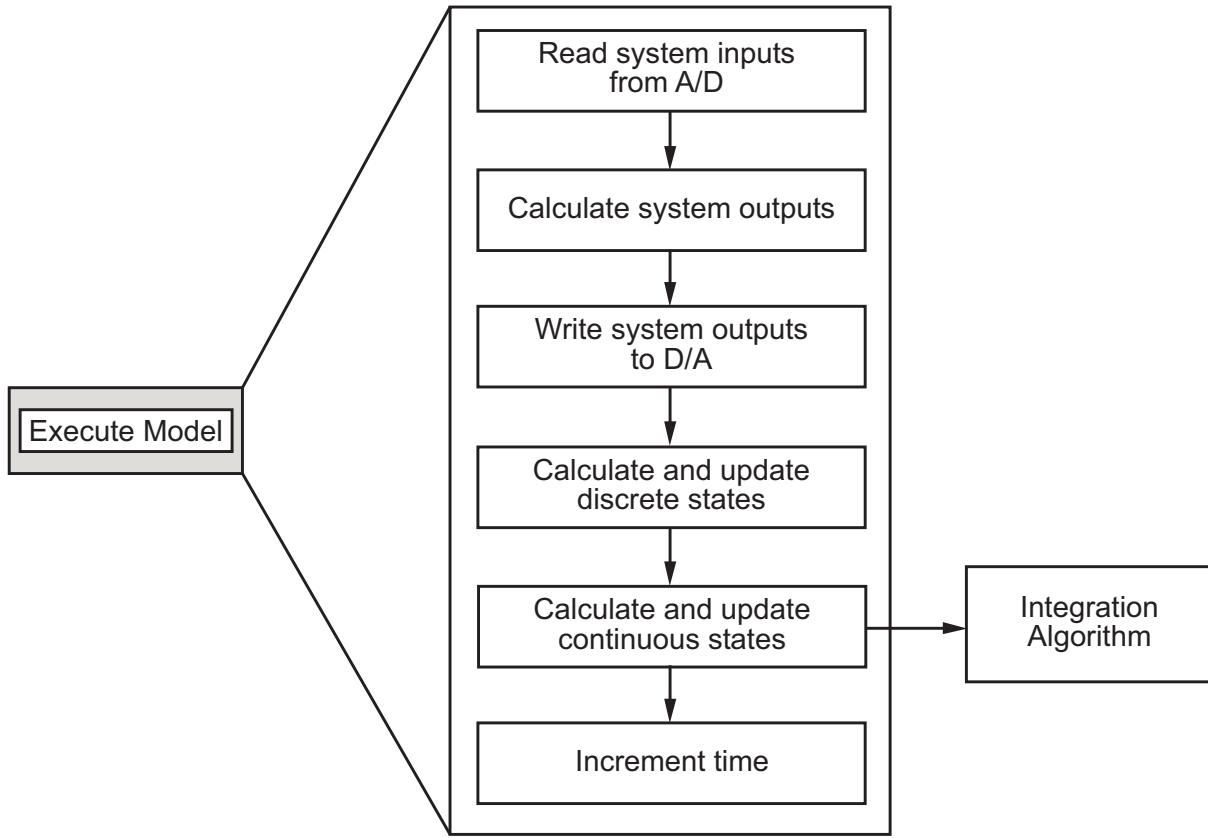
Rapid Prototyping System-Independent Components

These components are collectively called system independent because all environments use the same application modules to implement these operations. This section steps through the model code (and if the model has continuous states, calls one of the numerical integration routines). This section also includes the code that defines, creates, and destroys the real-time model data structure (`rtM`). The model code and all S-functions included in the program define their own `SimStructs`.

The model code execution driver calls the functions in the model code to compute the model outputs, update the discrete states, integrate the continuous states (if applicable), and update time. These functions then write their calculated data to the real-time model.

Model Execution. At each sample interval, the main program passes control to the model execution function, which executes one step though the model. This step reads inputs from the external hardware, calculates the model outputs, writes outputs to the external hardware, and then updates the states.

The next figure shows these steps.



Executing the Model

This scheme writes the system outputs to the hardware before the states are updated. Separating the state update from the output calculation minimizes the time between the input and output operations.

Integration of Continuous States. The real-time program calculates the next values for the continuous states based on the derivative vector, dx/dt , for the current values of the inputs and the state vector.

These derivatives are then used to calculate the next values of the states using a state-update equation. This is the state-update equation for the first-order Euler method (`ode1`)

$$x = x + \frac{dx}{dt} h$$

where h is the step size of the simulation, x represents the state vector, and dx/dt is the vector of derivatives. Other algorithms can make several calls to the output and derivative routines to produce more accurate estimates.

Note, however, that real-time programs use a fixed-step size because it is necessary to guarantee the completion of all tasks within a given amount of time. This means that, while you should use higher order integration methods for models with widely varying dynamics, the higher order methods require additional computation time. In turn, the additional computation time might force you to use a larger step size, which can diminish the improvement of accuracy initially sought from the higher order integration method.

Generally, the stiffer the equations, (that is, the more dynamics in the system with widely varying time constants), the higher the order of the method that you must use.

In practice, the simulation of very stiff equations is impractical for real-time purposes except at very low sample rates. You should test fixed-step size integration in the Simulink environment to check stability and accuracy before implementing the model for use in real-time programs.

For linear systems, it is more practical to convert the model that you are simulating to a discrete time version, for instance, using the `c2d` function in the Control System Toolbox™ product.

Application Modules for System-Independent Components. The system-independent components include these modules:

- `ode1.c`, `ode2.c`, `ode3.c`, `ode4.c`, `ode5.c`: These modules implement the integration algorithms supported for real-time applications. See “Choosing a Solver” in the Simulink documentation for more information about fixed-step solvers.
- `rt_sim.c`: Performs the activities necessary for one time step of the model. It calls the model function to calculate system outputs and then updates the discrete and continuous states.
- `simstruc_types.h`: Contains definitions of various events, including subsystem enable/disable and zero crossings. It also defines data-logging variables.

The system-independent components also include code that defines, creates, and destroys the real-time model data structure. All S-functions included in the program define their own **SimStructs**.

The **SimStruct** data structure encapsulates all the data relating to an S-function, including block parameters and outputs. See “The SimStruct” in the Writing S-Functions documentation for more information about **SimStruct**.

Rapid Prototyping Application Components

The application components contain the generated code for the Simulink model, including the code for any S-functions in the model. This code is referred to as the model code because these functions implement the Simulink model.

However, the generated code contains more than just functions to execute the model (as described in the previous section). There are also functions to perform initialization, facilitate data access, and complete tasks before program termination. To perform these operations, the generated code must define functions that

- Create the real-time model
- Initialize model size information in the real-time model
- Initialize a vector of sample times and sample time offsets and store this vector in the real-time model
- Store the values of the block initial conditions and program parameters in the real-time model
- Compute the block and system outputs
- Update the discrete state vector
- Compute derivatives for continuous models
- Perform an orderly termination at the end of the program (when the current time equals the final time, if a final time is specified)
- Collect block and scope data for data logging (either with the Real-Time Workshop product or third-party tools)

The Real-Time Model Data Structure. The real-time model data structure encapsulates model data and associated information necessary to fully describe the model. Its contents include

- Model parameters, inputs, and outputs
- Storage areas, such as dWork
- Timing information
- Solver identification
- Data logging information
- Simstructs for all child S-functions
- External mode information

The required information is stored in fields in the real-time model structure, which is defined in *model.h* as

```
/* Real-time Model Data Structure */
struct _rtModel_model_Tag {
    const char *path;
    const char *modelName;
    struct SimStruct_tag * *childSfunctions;
    const char *errorStatus;
    SS_SimMode simMode;
    RTWLogInfo *rtwLogInfo;
    RTWExtModeInfo *extModeInfo;
    RTWSolverInfo solverInfo;
    RTWSolverInfo *solverInfoPtr;
    void *sfcnInfo;

    /*
     * ModelData:
     * The following substructure contains information regarding
     * the data used in the model.
     */
    .
    .
}
```

The (possibly mangled) name of the model replaces *model* in the above tag. The individual substructures have been omitted, as they can vary.

For GRT targets, *model.h* also includes aliases to map global identifiers to identifiers used in prior versions (rtB, rtP, rtY, and so on). The following table lists the structure identifiers used in the generated code for these variants of the real-time model data structure. The column **GRT Symbol** contains the old-style (pre-Version 6) GRT identifiers, which are still used by the GRT calling interface, but not within the generated code.

Identifiers for Real-Time Model Data Structure Variants

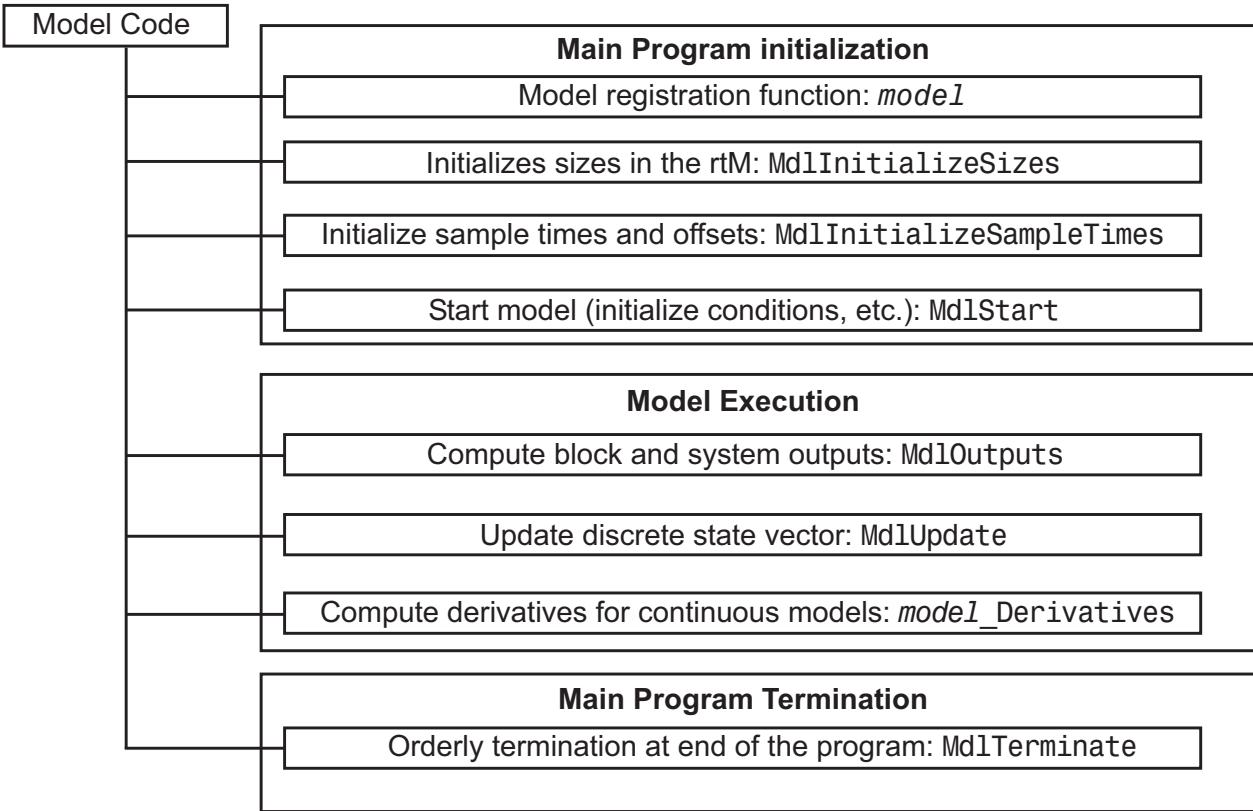
Identifier	GRT Symbol	Data
<i>model_B</i>	rtB	Block IO
<i>model_U</i>	rtU	External inputs
<i>model_X</i>	rtX	Continuous states
<i>model_Xdot</i>	rtXdot	State derivatives
<i>model_Xdix</i>	rtXdis	Continuous state disabled
<i>model_Y</i>	rtY	External outputs
<i>model_P</i>	rtP	Parameters
<i>rts</i>	rts	Child Simstruct
<i>model_DWork</i>	rtDWork	DWork
<i>model_ConstB</i>	rtC	Constant block IO define, structure
<i>model_ConstP</i>	rtcP	Constant parameter Structure
<i>model_PrevZCSigState</i>	rtPrevZCSigState	Previous zero-crossing signal states
<i>model_NonsampledZC</i>	rtNonsampledZC	Nonsampled zero-crossings

Real-Time Workshop Embedded Coder users can tailor identifiers, and can make them look like the GRT symbols listed above, should they desire such a coding style. The above GRT-ERT identifier equivalences (or at least as many of them as are required to build a given model) are established by using a set of `#define` macros in `model.h`, under the comment `/* Backward compatible GRT Identifiers */`.

The real-time model data structure is used for all targets. Prior to Version 5, the ERT target used the `rtObject` data structure, and other targets used the `Simstruct` data structure for encapsulating model data. Now all targets are treated the same, except for the fact that the real-time model data structure is *pruned* for ERT targets to save space in executables. Even when not pruned, the real-time model data structure is more space efficient than the root `Simstruct` used by earlier releases for non-ERT targets, as it only contains fields for child (S-function) `Simstructs` that are actually used in a model.

Rapid Prototyping Model Code Functions. The functions defined by the model code are called at various stages of program execution (that is, initialization, model execution, or program termination).

The next figure shows the functions defined in the generated code and shows what part of the program executes each function.



The Model Registration Function. The model registration function has the same name as the Simulink model from which it is generated. It is called directly by the main program during initialization. Its purpose is to initialize and return a pointer to the real-time model data structure.

Models Containing S-Functions. A *noninlined S-function* is any C or C++ MEX S-function that is not implemented using a customized TLC file. If you create a C or C++ MEX S-function as part of a Simulink model, it is by default noninlined unless you write your own TLC file that inlines it within the body of the *model.c* or *model.cpp* code. The Real-Time Workshop code generator automatically incorporates your noninlined C or C++ S-functions into the program if they adhere to the S-function API described in the Simulink documentation.

This format defines functions and a **SimStruct** that are local to the S-function. This allows you to have multiple instances of the S-function in the model. The

model's real-time model data structure contains a pointer to each S-function's `SimStruct`.

Code Generation and S-Functions. If a model contains S-functions, the source code for the S-function must be on the search path the `make` utility uses to find other source files. The directories that are searched are specified in the template makefile that is used to build the program.

S-functions are implemented in a way that is directly analogous to the model code. They contain their own public registration functions (called by the top model code) that initialize static function pointers in their `SimStructs`. When the top model needs to execute the S-function, it does so by using the function pointers in the S-function's `SimStruct`. There can be more than one S-function with the same name in your model. This is accomplished by having function pointers to static functions.

Inlining S-Functions. You can incorporate C/C++ MEX S-functions, along with the generated code, into the program executable. You can also write a target file for your C/C++ MEX S-function to inline the S-function, thus improving performance by eliminating function calls to the S-function itself. For more information on inlining S-functions, see the Target Language Compiler documentation.

Application Modules for Application Components. When the Real-Time Workshop software generates code, it produces the following files:

- `model.c` or `model.cpp`: C or C++ code generated from the Simulink block diagram. This code implements the block diagram's system equations as well as performing initialization and updating outputs.
- `model_data.c` or `model_data.cpp`: Optional file containing data for parameters and constant block I/O, which are also declared as `extern` in `model.h`. Only generated when `model_P` and `model_ConstB` structures are populated.
- `model_types.h`: Forward declarations for the real-time model data structure and the parameters data structure.
- `model.h`: Header file containing the block diagram's simulation parameters, I/O structures, work structures, and other declarations.

- *model_private.h*: Header file containing declarations of exported signals and parameters.

These files are named for the Simulink model from which they are generated.

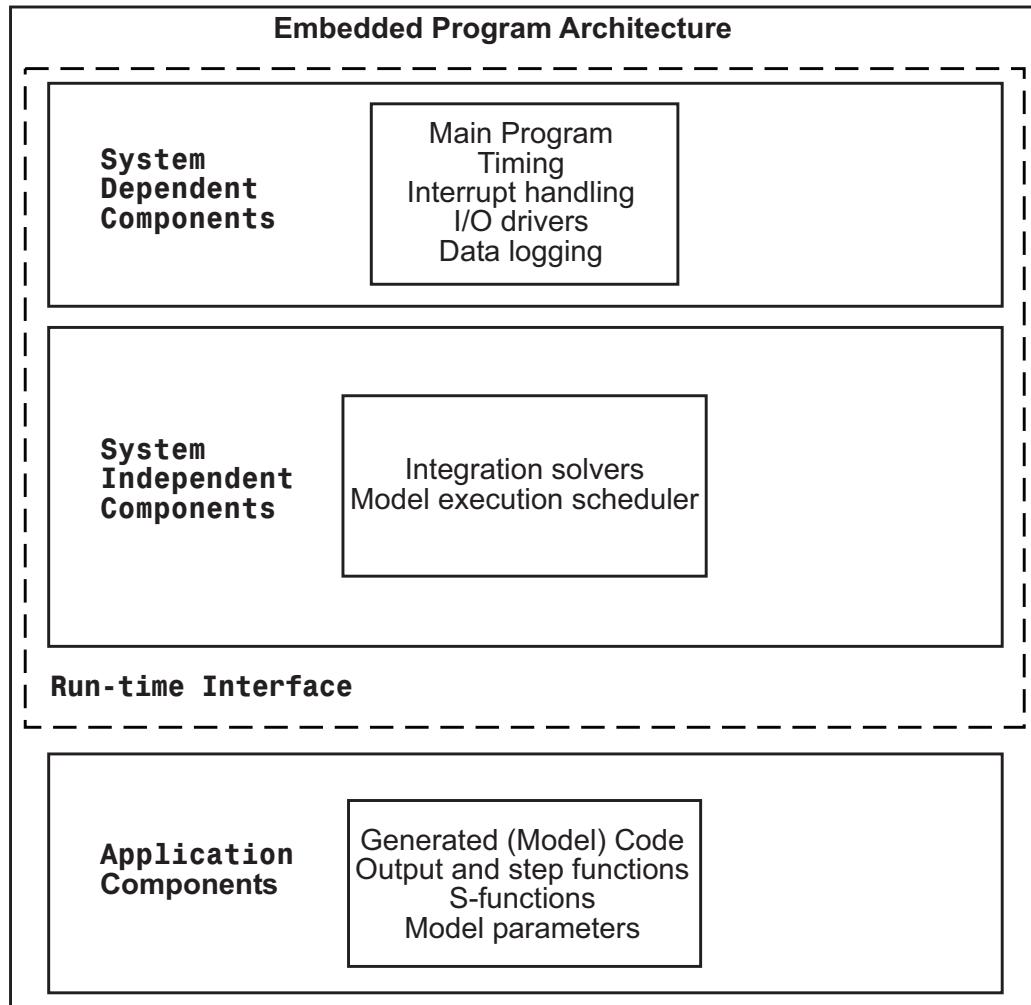
In addition, a dummy include file always named *rtmodel.h* is generated, which includes the above model-specific data structures and entry points. This enables the (static) target-specific main programs to reference files generated by the Real-Time Workshop code generator without needing to know the names of the models involved.

Another dummy file, *rtwtypes.h*, is generated, which simply includes *simstruc_types.h* (only for GRT and GRT malloc targets).

If you have created custom blocks using C/C++ MEX S-functions, you need the source code for these S-functions available during the build process.

Embedded Program Framework

The Real-Time Workshop Embedded Coder product provides a framework for embedded programs. Its architecture is outlined in the next figure.



Note the similarity between this architecture and the rapid prototyping architecture in the figure “Rapid Prototyping Program Architecture” on page 3-22. The main difference is the use of the `rtModel` data structure in place of the `SimStruct` data structure.

Using the previous figure, you can compare the embedded style of generated code, used in the Real-Time Workshop Embedded Coder product, with the rapid prototyping style of generated code of the previous section. Most of the rapid prototyping explanations in the previous section hold for the Real-Time Workshop Embedded Coder target. The Real-Time Workshop

Embedded Coder target simplifies the process of using the generated code in your custom-embedded applications by providing a model-specific API and eliminating the `SimStruct`. This target contains the same conceptual layering as the rapid prototyping target, but each layer has been simplified.

For a discussion of the structure of embedded real-time code, see the Real-Time Workshop Embedded Coder documentation.

Creating Subsystems

In this section...

“Introduction” on page 3-37

“Generating Code and Executables from Subsystems” on page 3-38

“Nonvirtual Subsystem Code Generation Options” on page 3-41

“Modularity of Subsystem Code” on page 3-50

Introduction

The Real-Time Workshop product allows you to control how code is generated for any nonvirtual subsystem. The categories of nonvirtual subsystems are:

- *Conditionally executed* subsystems: execution depends upon a control signal or control block. These include triggered subsystems, enabled subsystems, action and iterator subsystems, subsystems that are both triggered and enabled, and function call subsystems. See “Creating Conditional Subsystems” in the Simulink documentation for more information.
- *Atomic* subsystems: Any virtual subsystem can be declared atomic (and therefore nonvirtual) by using the **Treat as atomic unit** option in the Block Parameters dialog box.

Note You should declare virtual subsystems as atomic subsystems to ensure consistent simulation and execution behavior for your model. If you generate code for a virtual subsystem, the Real-Time Workshop software treats the subsystem as atomic and generates the code accordingly. The resulting code can change the execution behavior of your model, for example, by applying algebraic loops, and introduce inconsistencies with the simulation behavior.

See “Systems and Subsystems” in the Simulink documentation, and run the `s1_subsys_semantics` demo for more information on nonvirtual subsystems and atomic subsystems.

You can control the code generated from nonvirtual subsystems as follows:

- You can instruct the Real-Time Workshop code generator to generate separate functions, within separate code files if desired, for selected nonvirtual systems. You can control both the names of the functions and of the code files generated from nonvirtual subsystems.
- You can cause multiple instances of a subsystem to generate *reusable* code, that is, as a single reentrant function, instead of replicating the code for each instance of a subsystem or each time it is called.
- You can generate inlined code from selected nonvirtual subsystems within your model. When you inline a nonvirtual subsystem, a separate function call is not generated for the subsystem.

Generating Code and Executables from Subsystems

The Real-Time Workshop software can generate code and build an executable from any subsystem within a model. The code generation and build process uses the code generation and build parameters of the root model.

To generate code and build an executable from a subsystem,

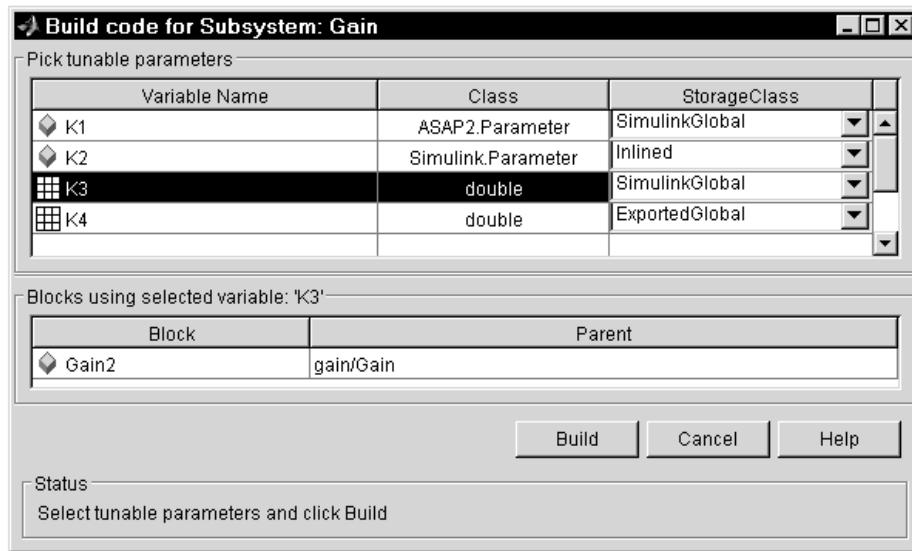
- 1 Set up the desired code generation and build parameters in the Configuration Parameters dialog box, just as you would for code generation from a model.
- 2 Select the desired subsystem block.
- 3 Right-click the subsystem block and select **Build Subsystem** from the **Real-Time Workshop** submenu of the subsystem block's context menu.

Alternatively, you can select **Build Subsystem** from the **Real-Time Workshop** submenu of the **Tools** menu. This menu item is enabled when a subsystem is selected in the current model.

Note If the model is operating in external mode when you select **Build Subsystem**, the Real-Time Workshop build process automatically turns off external mode for the duration of the build, then restores external mode upon its completion.

- 4** The **Build Subsystem** window opens. This 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 all the blocks that reference the parameter and the parent system of each such block.

The **StorageClass** column contains a popup menu for each row. The menu lets you set the storage class of any parameter or inline the parameter. To inline a parameter, select the **Inline** option from the menu. To declare a parameter to be tunable, set the storage class to any value other than **Inline**.



In the previous figure, the parameter **K2** is inlined, while the other parameters are tunable and have various storage classes.

See Chapter 10, “Parameter Considerations” and Chapter 12, “Simulink Data Object Considerations” for more information on tunable and inlined parameters and storage classes.

- 5** After selecting tunable parameters, click the **Build** button. This initiates the code generation and build process.

- 6** The build process displays status messages in the MATLAB Command Window. When the build completes, the generated executable is in your working directory. The name of the generated executable is *subsystem.exe* (on PC platforms) or *subsystem* (on The Open Group UNIX® platforms), where *subsystem* is the name of the source subsystem block.

The generated code is in a build subdirectory, named *subsystem_target_rtw*, where *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 **Tools > Real-Time Workshop > Generate S-function**, or you can use a right-click subsystem build. See “Automated S-Function Generation” on page 3-104 and “Generating S-Function Wrappers” for more details.

Real-Time Workshop Subsystem Build Limitations

The following limitations apply to building subsystems using the Real-Time Workshop software:

- When you right-click build a subsystem that includes an Outport block for which the signal specification **Specify properties via bus object** is selected, the Real-Time Workshop build process requires that you set the **Signal label mismatch** option on the **Diagnostics > Connectivity** pane of the Configuration Parameters dialog box for the parent model to **error**. You need to address any errors that occur by properly setting signal labels.
- 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 > Solver** pane, set:
 - a. **Type** to **Fixed-step**.
 - b. **Periodic sample time constraint** to **Ensure sample time independent**.
 - c. Click **Apply**.
 - 1 Update the model. If the model is sample-time dependent, Simulink generates an error in the process of updating the diagram.

Nonvirtual Subsystem Code Generation Options

For any nonvirtual subsystem, you can choose the following code generation options from the **Real-Time Workshop system code** menu in the subsystem Block parameters dialog box:

- **Auto**: This is the default option, and provides the greatest flexibility in most situations. See “Auto Option” on page 3-41 below.
- **Inline**: This option explicitly directs the Real-Time Workshop code generator to inline the subsystem unconditionally.
- **Function**: This option explicitly directs the Real-Time Workshop code generator to generate a separate function with no arguments, and (optionally), place the subsystem in a separate file. You can name the generated function and file. As functions created with this option rely on global data, they are not reentrant.
- **Reusable function**: Generates a function with arguments that allows the subsystem’s code to be shared by other instances of it in the model. To enable sharing, the Real-Time Workshop software must be able to determine (by using checksums) that subsystems are identical. The generated function will have arguments for block inputs and outputs (`rtB_*`), continuous states (`rtDW_*`), parameters (`rtP_*`), and so on.

Note You should not directly call reusable functions generated by the Real-Time Workshop product. The call interface is subject to change.

The following sections discuss these options further.

Auto Option

The **Auto** option is the default, and is generally appropriate. **Auto** causes the Real-Time Workshop code generator to inline the subsystem when there is only one instance of it in the model. When multiple instances of a subsystem exist, the **Auto** option results in a single copy of the function whenever possible (as a reusable function). Otherwise, the result is as though you selected **Inline** (except for function call subsystems with multiple callers, which is handled as if you specified **Function**). Choose **Inline** to always inline subsystem code, or **Function** when you specifically want to generate

a separate function without arguments for each instance, optionally in a separate file.

Note 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**. It is best to use one or the other, as using both creates two reusable functions, one for each designation. The outcomes of these choices differ only when reuse is not possible.

To use the **Auto** option,

- 1 Select the subsystem block. Then select **Subsystem Parameters** from the Simulink model editor **Edit** menu. The Block Parameters dialog box opens, as shown in the next figure.

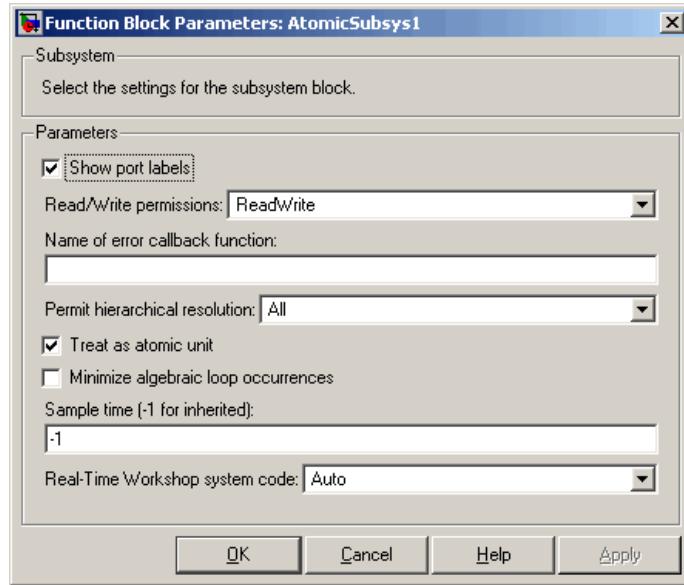
Alternatively, you can open the Block Parameters dialog box by

- Shift-double-clicking the subsystem block
- Right-clicking the subsystem block and selecting **Block parameters** from the menu

- 2 If the subsystem is virtual, select **Treat as atomic unit** as shown in the next figure. This makes the subsystem nonvirtual, and the **Real-Time Workshop system code** option becomes enabled.

If the system is already nonvirtual, the **Real-Time Workshop system code** option is already enabled.

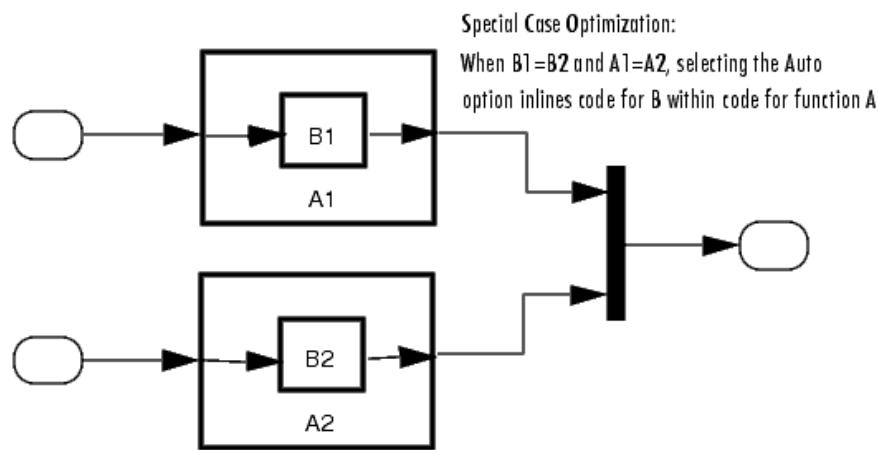
- 3 Select **Auto** from the **Real-Time Workshop system code** menu as shown in the figure below.



4 Click **Apply** and close the dialog box.

The border of the subsystem thickens, indicating that it is nonvirtual.

Auto Optimization for Special Cases. Rather than reverting to **Inline**, the **Auto** option can optimize code in special 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 with the Auto Option](#) on page 3-44, contains identical subsystems A1 and A2. A1 contains subsystem B1, and A2 contains subsystem B2, which are themselves identical. In such cases, the **Auto** option causes one function to be generated which is called for both A1 and A2, and this function contains one piece of inlined code to execute B1 and B2, ensuring that the resulting code will run as efficiently as possible.



Reuse of Identical Nested Subsystems with the Auto Option

Inline Option

As noted above, you can choose to inline subsystem code when the subsystem is nonvirtual (virtual subsystems are always inlined).

Exceptions to Inlining. There are certain cases in which the Real-Time Workshop code generator does not inline a nonvirtual subsystem, even though the **Inline** option is selected. These cases are

- 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 such 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 Real-Time Workshop code generator forces one of the subsystems to be generated as a function instead of inlining it. The product selects the subsystem to be generated as a function based on the order in which the subsystems are sorted internally.
- 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. This might be the case when user-defined Asynchronous Interrupt blocks or Task Synchronization blocks are required. Such blocks must be generated

as functions. The Asynchronous Interrupt and Task Synchronization blocks, located in the VxWorks® block library shipped with the Real-Time Workshop product, use the `SS_OPTION_FORCE_NONINLINED_FCNCALL` option.²

To generate inlined subsystem code,

- 1 Select the subsystem block. Then select **Subsystem Parameters** from the Simulink model editor **Edit** menu. The Block Parameters dialog box opens.

Alternatively, you can open the Block Parameters dialog box by

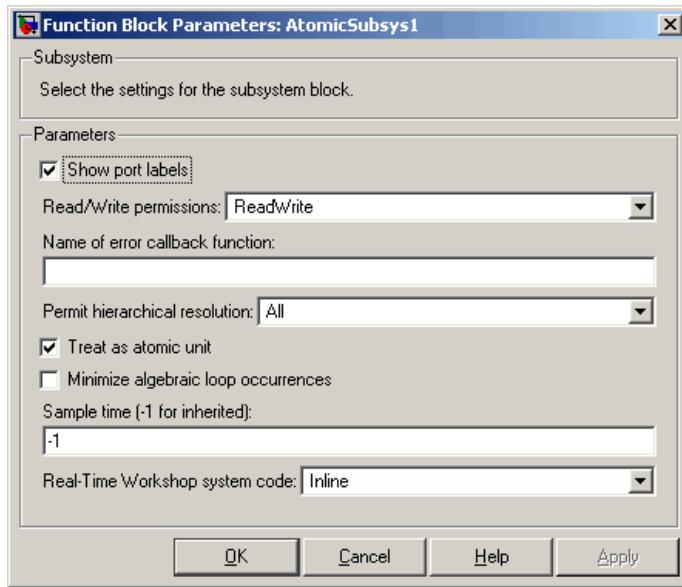
- Shift-double-clicking the subsystem block
- Right-clicking the subsystem block and selecting **Block parameters** from the menu

- 2 If the subsystem is virtual, select **Treat as atomic unit** as shown in the next figure. This makes the subsystem atomic, and the **Real-Time Workshop system code** menu becomes enabled.

If the system is already nonvirtual, the **Real-Time Workshop system code** menu is already enabled.

- 3 Select **Inline** from the **Real-Time Workshop system code** menu as shown in the figure below.

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



4 Click **Apply** and close the dialog box.

When you generate code from your model, the Real-Time Workshop code generator writes inline code within *model.c* or *model.cpp* (or in its parent system's source file) to perform subsystem computations. You can identify this code by system/block identification tags, such as the following.

```
/* Atomic SubSystem Block: <Root>/AtomicSubsys1 */
```

See “Tracing Generated Code” on page 43-3 for more information on system/block identification tags.

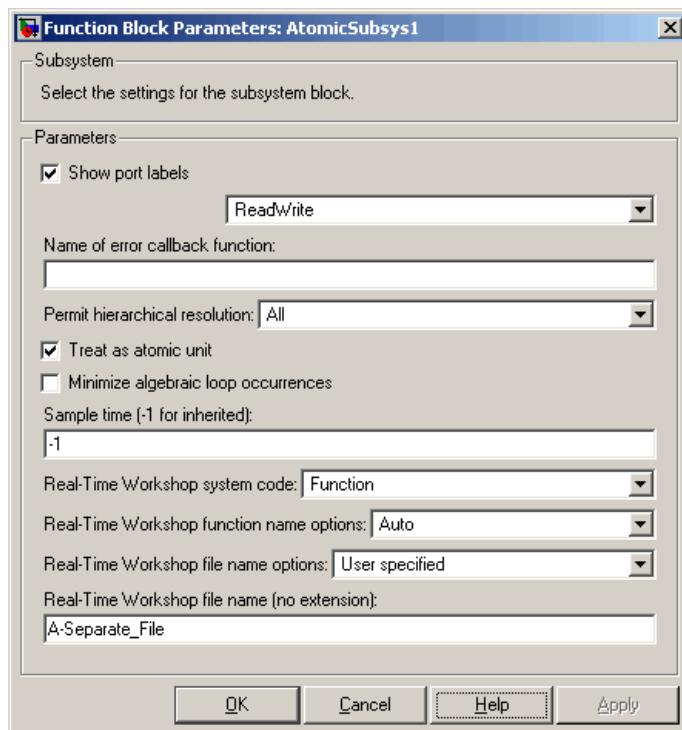
Function Option

Choosing the **Function** or **Reusable function** option lets you direct the Real-Time Workshop code generator to generate a separate function and optionally a separate file for the subsystem. When you select the **Function** option, two additional options are enabled:

- The **Real-Time Workshop function name options** menu lets you control the naming of the generated function.

- The **Real-Time Workshop file name options** menu lets you control the naming of the generated file (if a separate file is generated and you select the **User specified** option).

The figure below shows the Block Parameters dialog box with the **Function** option selected, with **Real-Time Workshop file name options** set to **User specified**, and with a name specified for the generated file.



Subsystem Function Code Generation Option with User-Specified File Name

Real-Time Workshop Function Name Options Menu. This menu offers the following choices, but the resulting identifiers are also affected by which General code appearance options are in effect for the model:

- Auto:** By default, the Real-Time Workshop code generator assigns a unique function name using the default naming convention: `model_subsystem()`, where `subsystem` is the name of the subsystem (or that of an identical one when code is being reused).

- **Use subsystem name:** the Real-Time Workshop code generator uses the subsystem name as the function name.

Note When a subsystem is a library block, the **Use subsystem name** option causes its function identifier (and file name, see below) to be that of the library block, regardless of the names used for that subsystem in the model.

- **User specified:** When this option is selected, the **Real-Time Workshop function name** field is enabled. Enter any legal C or C++ function name (which must be unique).

Real-Time Workshop File Name Options Menu. This menu offers the following choices:

- **Use subsystem name:** the Real-Time Workshop software generates a separate file, using the subsystem (or library block) name as the file name.

Note When a subsystem's **Real-Time Workshop file name options** is set to **Use subsystem name**, the subsystem file name is mangled if the model contains Model blocks, or if a model reference target is being generated for the model. In these situations, the file name for the subsystem consists of the subsystem name prefixed by the model name.

- **Use function name:** the Real-Time Workshop software generates a separate file, using the function name (as specified by the **Real-Time Workshop function name** options) as the file name.
- **User specified:** When this option is selected, the **Real-Time Workshop file name (no extension)** text entry field is enabled. The Real-Time Workshop software generates a separate file, using the name you enter as the file name. Enter any file name, but do not include the .c or .cpp (or any other) extension. This file name need not be unique.

Note While a subsystem source file name need not be unique, you must avoid giving nonunique names that result in cyclic dependencies (for example, `sys_a.h` includes `sys_b.h`, `sys_b.h` includes `sys_c.h`, and `sys_c.h` includes `sys_a.h`).

- **Auto:** The Real-Time Workshop software does *not* generate a separate file for the subsystem. Code generated from the subsystem is generated within the code module generated from the subsystem's parent system. If the subsystem's parent is the model itself, code generated from the subsystem is generated within `model.c` or `model.cpp`.

To generate both a separate subsystem function and a separate file,

- 1 Select the subsystem block. Then select **Subsystem Parameters** from the Simulink model editor **Edit** menu, to open the Block Parameters dialog box.

Alternatively, you can open the Block Parameters dialog box by

- Shift-double-clicking the subsystem block
- Right-clicking the subsystem block and selecting **Block parameters** from the menu.

- 2 If the subsystem is virtual, select **Treat as atomic unit**. The **Real-Time Workshop system code** menu becomes enabled.

If the system is already nonvirtual, the **Real-Time Workshop system code** menu is already enabled.

- 3 Select **Function** from the **Real-Time Workshop system code** menu as shown in Subsystem Function Code Generation Option with User-Specified File Name on page 3-47.

- 4 Set the function name, using the **Real-Time Workshop function name** options described in “Real-Time Workshop Function Name Options Menu” on page 3-47.

- 5 Set the file name, using any **Real-Time Workshop file name** option other than **Auto** (options are described in “Real-Time Workshop File Name Options Menu” on page 3-48).

Subsystem Function Code Generation Option with User-Specified File Name on page 3-47 shows the use of the `User Specified` file name option.

- 6 Click **Apply** and close the dialog box.

Modularity of Subsystem Code

Code generated from nonvirtual subsystems, when written to separate files, is not completely independent of the generating model. For example, subsystem code may reference global data structures of the model. Each subsystem code file contains appropriate include directives and comments explaining the dependencies. The Real-Time Workshop 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” on page 22-2.

Creating Model Components

In this section...

- “Introduction” on page 3-51
- “Overview of Referenced Model Code Generation” on page 3-51
- “Project Directory Structure for Model Reference Targets” on page 3-53
- “Configuring Referenced Models” on page 3-54
- “Building Model Reference Targets” on page 3-54
- “Real-Time Workshop Model Referencing Requirements” on page 3-55
- “Storage Classes for Signals Used with Model Blocks” on page 3-62
- “Inherited Sample Time for Referenced Models” on page 3-66
- “Customizing the Library File Suffix, Including the File Type Extension” on page 3-68
- “Real-Time Workshop Model Referencing Limitations” on page 3-68

Introduction

This section describes model referencing considerations that apply specifically to code generation by the Real-Time Workshop software with GRT and ERT system targets. This section assumes that you understand referenced models and their terminology and requirements, as described in “Referencing a Model”. This section does not repeat information that appears in that chapter.

Overview of Referenced Model Code Generation

When generating code for a referenced model hierarchy, the Real-Time Workshop software generates 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 as needed to compute the referenced model outputs. Model reference targets are sometimes called *Real-Time Workshop targets*.

Be careful not to confuse a model reference target (Real-Time Workshop target) with any of these other types of targets:

- Hardware target — A platform for which the Real-Time Workshop software generates code
- System target — A file that tells the Real-Time Workshop software how to generate code for particular purpose
- Rapid Simulation target (RSim) — A system target file supplied with the Real-Time Workshop product
- Simulation target — A MEX-file that implements a referenced model that executes with Simulink® Accelerator™ software

The Real-Time Workshop code generator places the code for the top model of a hierarchy in the current working directory, and the code for submodels in a directory named `s1prj` within the current working directory. Subdirectories in `s1prj` provide separate places for different types of files. See “Project Directory Structure for Model Reference Targets” on page 3-53 for details.

By default, the product uses *incremental code generation*. When generating code, it compares the date, and optionally, the structure of referenced model files with the generated code files to determine whether it is necessary to regenerate model reference targets. You can also force or prevent code generation by using a diagnostic setting **Configuration Parameters > Model Referencing > Rebuild options**.

In addition to incremental code generation, the Real-Time Workshop 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 needed.

Most code generation considerations are the same whether or not a model includes any referenced models: the Real-Time Workshop 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.

Custom targets must declare themselves to be model reference compliant if they need to support Model blocks. See “Supporting Model Referencing” on page 21-110 for details.

Referenced Model Code Generation Tutorial

You can get hands-on experience with creating referenced models and generating code for them by working through the model reference tutorial. See “Generating Code for a Referenced Model”.

Project Directory Structure for Model Reference Targets

Code for models referenced by using Model blocks is generated in project directories within the current working directory. The top-level project directory is always named `s1prj`. The next level within `s1prj` contains parallel build area subdirectories.

The following table lists principal project directories 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).

Directories and Files	Description
<code>s1prj/sim/model/</code>	Simulation target files for referenced models
<code>s1prj/sim/model/tmwinternal</code>	MAT-files used during code generation
<code>s1prj/target/model/referenced_model_includes</code>	Header files from models referenced by this model
<code>s1prj/target/model</code>	Model reference target files
<code>s1prj/target/model/tmwinternal</code>	MAT-files used during code generation
<code>s1prj/s1_proj.tmw</code>	Marker file
<code>s1prj/target/_sharedutils</code>	Utility functions for model reference targets, shared across models
<code>s1prj/sim/_sharedutils</code>	Utility functions for simulation targets, shared across models

If you are building code for more than one referenced model within the same working directory, model reference files for all such models are added to the existing `s1prj` directory.

Configuring 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 “Configuring the Hardware Implementation” on page 33-2 in the Real-Time Workshop documentation for information on how this option affects code generation. For more information, see “Model Blocks and Direct Feedthrough”.

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 Lookup Table (n-D) 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 “Rounding” in the *Simulink Fixed Point User’s Guide*.

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 “Referencing a Model” in the Simulink documentation.

Building Model Reference Targets

By default, the Simulink engine rebuilds simulation targets as needed before the Real-Time Workshop software generates model reference targets. You can change the rebuild criteria or specify that the engine always or never rebuilds targets. See “Rebuild options” for details.

The Real-Time Workshop software generates a model reference target directly from the Simulink model. The product automatically generates or regenerates model reference targets as needed.

You can command the Simulink and Real-Time Workshop products to generate a simulation target for an Accelerator mode referenced model, and a model reference target for any referenced model, by executing the `slbuild` command with appropriate arguments in the MATLAB Command Window.

The Real-Time Workshop software generates only one model reference target for all instances of a referenced model. See “Reusable Code and Referenced Models” on page 3-73 for details.

Reducing Change Checking Time

You can reduce the time that the Simulink and Real-Time Workshop products spend checking whether any or all simulation targets and model reference targets need to be rebuilt by setting configuration parameter values as follows:

- In the top model, set **Configuration Parameters > Model Referencing > Rebuild options** to If any changes in known dependencies detected. (See “Rebuild options”.)
- In all referenced models throughout the hierarchy, set **Configuration Parameters > 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, so setting either value for any instance of a referenced model sets it for all instances of that model.

Real-Time Workshop Model Referencing Requirements

A model reference hierarchy must satisfy various Real-Time Workshop requirements, as described in this section. In addition to these requirements, a model referencing hierarchy to be processed by the Real-Time Workshop 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 Real-Time Workshop limitations listed in “Real-Time Workshop Model Referencing Limitations” on page 3-68

Configuration Parameter Requirements

A referenced model uses a configuration set in the same way that any other model does, as described in “Setting Up Configuration Sets”. 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 Real-Time Workshop software to an inconsistent or unusable configuration parameter depends on the parameter:

- Where an inconsistency has no significance, or a trivial resolution exists that carries no risk, 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.
- Where no acceptable resolution is possible, the product generates an error. You must then change some or all parameter values to eliminate the problem.

When a model reference hierarchy contains many submodels that have incompatible parameter values, or a changed parameter value must propagate to many submodels, 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 “Referencing Configuration Sets” 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 Real-Time Workshop 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	Emulation hardware options	All values must be the same for top and referenced models.
Real-Time Workshop	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.

Configuration Requirements for Model Referencing with All System Targets (Continued)

Dialog Box Pane	Option		Requirement
Interface	Target function library		Must be the same for top and referenced models.
	Data exchange > Interface	C API	The C API check boxes for signals, parameters, and 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 Real-Time Workshop software temporarily sets it to off during code generation.

Configuration Requirements for Model Referencing with ERT System Targets

Dialog Box Pane	Option	Requirement
Real-Time Workshop	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.

Configuration Requirements for Model Referencing with ERT System Targets (Continued)

Dialog Box Pane	Option	Requirement
Interface	Support floating-point numbers	If off for top model, must be off for referenced models.
	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.
	Terminate function required	Must be the same for top and referenced models.
	Suppress error status in real-time model	If on for top model, must be on for referenced models.
Templates	Target operating system	Must be the same for top and referenced models.

Configuration Requirements for Model Referencing with ERT System Targets (Continued)

Dialog Box Pane	Option	Requirement
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, there can be no collisions between the names of the constituent models. 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 (if needed). 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.

Real-Time Workshop Embedded Coder Naming Requirements. The Real-Time Workshop Embedded Coder product provides a **Symbol format** field that 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).
- The **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.

See “Real-Time Workshop Pane: Symbols” and “Preparing Models for Code Generation” for more information.

Custom Target Requirements

A custom target must meet various requirements in order to support model referencing. See “Supporting Model Referencing” on page 21-110 for details.

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 any 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 any exported signal, the top mode is responsible for defining (allocating memory for) the signal, whether or not the top model itself uses the signal.
- All 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

All 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 any 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).
- All global parameters are accessed directly (as global memory). They are not passed as arguments to any of the functions that are generated for any of 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 “Parameterizing Model References” in the Simulink documentation for specific details.

Effects of Signal Name Mismatches

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 import or outport within that referenced model. Because referenced models are compiled independently without regard to any parent model, they cannot adapt to all possible variations in how parent models label and store signals.

The Real-Time Workshop 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 Between Parent and Referenced Models

Referenced Model	Parent Model	Signal Passing Method	Signal Mismatch Checking
Auto	Any	Function argument	None
SimulinkGlobal or resolved to Signal Object	Any	Function argument	Label Mismatch Diagnostic (none / warning / error)
Global	Auto or SimulinkGlobal	Global variable	Label Mismatch Diagnostic (none / warning / error)
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 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 Mismatch** diagnostic is applied.

You can set the **Signal Mismatch** diagnostic to error, warning, or none in the **Configuration Parameters > Diagnostics > Connectivity** dialog.

Inherited Sample Time for Referenced Models

See “Inheriting 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 demo `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);

```

Customizing the Library File Suffix, Including the File Type Extension

You can control the library file suffix, including the file type extension, that the Real-Time Workshop code generator uses to name generated model reference libraries by specifying the string for the suffix with the model configuration parameter **TargetLibSuffix**. The string must include a period (.). If you do not set this parameter,

On a...	The Real-Time Workshop Software Names the Libraries...
Microsoft Windows® system	<i>model_rtwlib.lib</i>
The Open Group UNIX system	<i>model_rtwlib.a</i>

Real-Time Workshop Model Referencing Limitations

The following Real-Time Workshop 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 “Simulink Model Referencing Limitations”.
- The Real-Time Workshop requirements applicable to the code generation target, as listed in “Configuration Parameter Requirements” on page 3-56.

Customization Limitations

- The Real-Time Workshop code generator ignores custom code settings in the **Configuration Parameter** dialog box and custom code blocks when generating code for a referenced model.

- Some restrictions exist on grouped custom storage classes in referenced models. See “Custom Storage Class Limitations” for details.
- 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.

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 Real-Time Workshop 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 Real-Time Workshop software disables the option before code generation and re-enables it afterwards.

State Initialization Limitation

When a top model uses the **Load from workspace > Initial state** option to specify initial conditions, Real-Time Workshop does not initialize the states of any referenced models.

Reusability Limitations

If a referenced model used for code generation has any of the following properties, the model must specify **Configuration Parameters > 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 the parameter is not set correctly, 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.
- The Real-Time Workshop software cannot generate code for a referenced model that includes noninlined S-functions.
- A referenced model cannot use noninlined S-functions generated by the Real-Time Workshop software.
- The Real-Time Workshop S-function target does not support 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 Real-Time Workshop 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 **Tools > Real-Time Workshop > Build 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 can affect code reuse, as described in “Reusable Code and Referenced Models” on page 3-73.

Target Limitations

- Real-Time Workshop `grt_malloc` targets do not support model reference.

- The Real-Time Workshop 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. See “Model Blocks and Direct Feedthrough” for details.
- The **External mode** option is not supported. If it is enabled, it is ignored during code generation.

Creating Reusable Components

In this section...

- “Reusable Function Option” on page 3-73
- “Reusable Code and Referenced Models” on page 3-73
- “Generating Reusable Code from Stateflow Charts” on page 3-77
- “Generating Reusable Code for Subsystems Containing S-Function Blocks” on page 3-77
- “Code Reuse Limitations” on page 3-79
- “Determining Why Subsystem Code Is Not Reused” on page 3-80

Reusable Function Option

The difference between functions and reusable functions is that the latter have data passed to them as arguments (enabling them to be reentrant), while the former communicate by using global data. Choosing the **Reusable function** option directs the Real-Time Workshop code generator to generate a single function (optionally in a separate file) for the subsystem, and to call that code for each identical subsystem in the model, if possible.

Note The **Reusable function** option yields code that is called from multiple sites (hence reused) only when the **Auto** option would also do so. The difference between these options' behavior is that when reuse is not possible, selecting **Auto** yields inlined code (or if circumstances prohibit inlining, creates a function without arguments), while choosing **Reusable function** yields a separate function (with arguments) that is called from only one site.

For a summary of code reuse limitations, see “Code Reuse Limitations” on page 3-79.

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 Import 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 “Creating Subsystems” on page 3-37. However, the following restrictions apply to such builds:

- ERT S-functions do not support subsystems that contain a continuous sample time.
- The Real-Time Workshop 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.

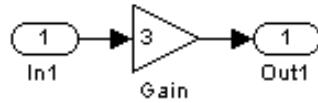
Note You can generate subsystem code using any target configuration available in the System Target File Browser. However, if the S-function target is selected, **Build Subsystem** behaves identically to **Generate S-function**. (See “Automated S-Function Generation” on page 3-104.)

Code Reuse and Model Blocks with Root Import or Outport Blocks

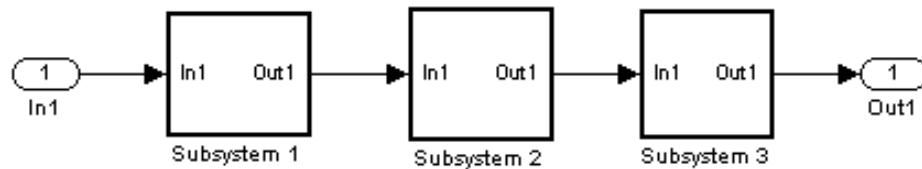
Reusable functions with inputs or outputs connected to a referenced model's root Import or Outport block can affect code reuse. This means that code for certain atomic subsystems cannot be reused in a model reference context the same way it is reused in a standalone model.

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**
- Set **Real-Time Workshop system code** 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 Real-Time Workshop 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` project directory), 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' */

```

```

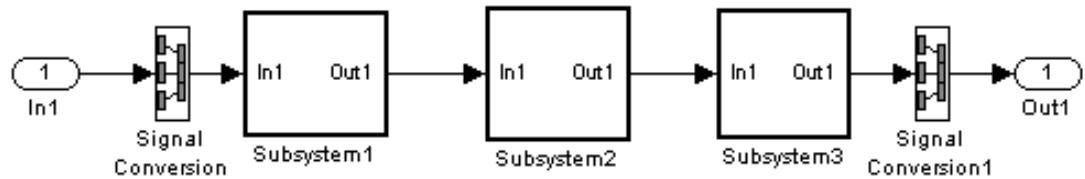
    localB->Gain_w = (*rtu_0) * 3.0;
}

/* 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 — and therefore assure code reuse — is to insert Signal Conversion blocks. Place one between the Import 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 Import 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.

Generating Reusable Code from Stateflow Charts

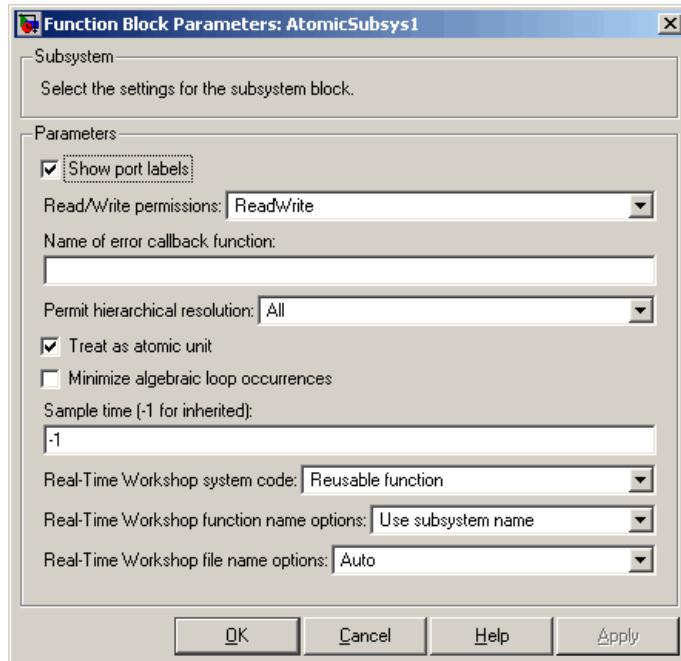
You can generate reusable code from a Stateflow chart, or from a subsystem containing a chart, *except* in the following cases:

- The Stateflow chart contains exported graphical functions.
- The Stateflow model contains machine parented events.

Generating Reusable Code for Subsystems Containing S-Function Blocks

Regarding S-Function blocks, there are several requirements that need to be met in order for subsystems containing them to be reused. See “Writing S-Functions That Support Code Reuse” on page 32-81 for the list of requirements.

When you select the **Reusable function** option, two additional options are enabled, **Real-Time Workshop function name options** and **Real-Time Workshop file name options**. See the explanation of “Function Option” on page 3-46 for descriptions of these options and fields. 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 Real-Time Workshop software to be able to reuse the subsystem code.



Subsystem Reusable Function Code Generation Option

To request that the Real-Time Workshop software generate reusable subsystem code,

- 1 Select the subsystem block. Then select **Subsystem Parameters** from the Simulink model editor **Edit** menu. The Block Parameters dialog box opens.

Alternatively, you can open the Block Parameters dialog box by:

- Shift-double-clicking the subsystem block
- Right-clicking the subsystem block and selecting **Block parameters** from the menu.

- 2 If the subsystem is virtual, select **Treat as atomic unit**. The **Real-Time Workshop system code** menu becomes enabled.

If the system is already nonvirtual, the **Real-Time Workshop system code** menu is already enabled.

3 Select **Reusable function** from the **Real-Time Workshop system code** menu as shown in Subsystem Reusable Function Code Generation Option on page 3-78.

4 If you want to give the function a specific name, set the function name, using the **Real-Time Workshop function name** options described in “Real-Time Workshop Function Name Options Menu” on page 3-47.

If you do not choose the **Real-Time Workshop function name** Auto option, and want code to be reused, you must assign exactly the same function name to all other subsystem blocks that you want to share this code.

5 If you want to direct the generated code to a specific file, set the file name using any **Real-Time Workshop file name** option other than Auto (options are described in “Real-Time Workshop File Name Options Menu” on page 3-48).

In order for code to be reused, you must repeat this step for all other subsystem blocks that you want to share this code, using the same file name.

6 Click **Apply** and close the dialog box.

Code Reuse Limitations

The Real-Time Workshop software uses a checksum to determine whether subsystems are identical. You cannot reuse subsystem code if:

- Multiple ports of a subsystem share the same source.
- 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.

Some of these situations can arise even when you copy and paste subsystems within or between models or you construct them manually such that they are identical. If you select **Reusable function** and the Real-Time Workshop software determines that code for a subsystem cannot be reused, it generates a separate function that is not reused. The code generation report can 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, you choose **Auto** for the **Real-Time Workshop system code** option.

Use of the following blocks in a subsystem can also prevent its code from being reused:

- Scope blocks (with data logging enabled)
- S-Function blocks that fail to meet certain criteria (see “Writing S-Functions That Support Code Reuse” on page 32-81)
- To File blocks (with data logging enabled)
- To Workspace blocks (with data logging enabled)

Determining Why Subsystem Code Is Not Reused

Due to the limitations noted in “Code Reuse Limitations” on page 3-79, the Real-Time Workshop software might not reuse generated code as you expect. To determine why code generated for a subsystem is not reused,

- 1 Review the Subsystems section of the HTML code generation report
- 2 If you cannot determine why based on the report, compare subsystem checksum data

Reviewing the Subsystems Section of the HTML Code Generation Report

If you determine that the Real-Time Workshop 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 “Generating a Report” on page 23-2). The Subsystems section contains

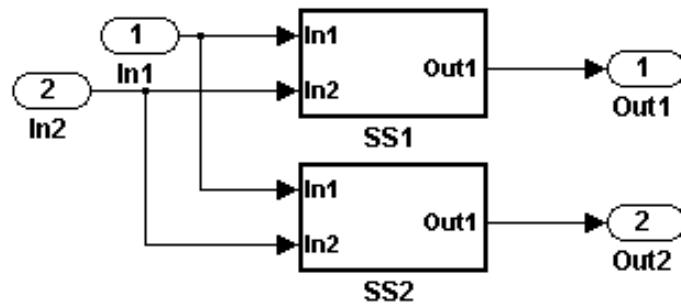
- A table that summarizes how nonvirtual subsystems were converted to generated code
- Diagnostic information that explains why the contents of some subsystems were not generated as reusable code

In addition to diagnosing exceptions, the Subsections 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 `rtdemo_atomic` demo model.

Comparing Subsystem Checksum Data

If the HTML code generation report indicates that no code reuse exceptions occurred and code for a subsystem you expect to be reused is not reused, you can determine why by accessing and comparing subsystem checksum data. The Real-Time Workshop software determines whether subsystems are identical by comparing subsystem checksums, as noted in “Code Reuse Limitations” on page 3-79.

Consider the demo model, `rtdemo_ssreuse`.



SS1 and **SS2** are instances of the same subsystem, and in both instances the subsystem parameter **Real-Time Workshop system code** is set to **Reusable function**.

The following example demonstrates how to use the method `Simulink.SubSystem.getChecksum` to get the checksum for a subsystem and compare the results to determine why code is not reused.

- 1 Open the model `rtwdemo_ssreuse` and save a copy of the demo in a directory where you have write access.
- 2 Select subsystem **SS1** in the model window and in the command window enter

```
SS1 = gcb;
```

- 3 Select subsystem **SS2** in the model window and 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 by right-clicking the subsystem and selecting **Look Under Mask** in the context menu. A block diagram of the subsystem appears.
 - b** Double-click the Lookup Table block to open the Block Parameters dialog box.
 - c** Click **Signal Attributes**.
 - d** Select **int8** for **Output data type** and click **OK**.
- 7** Get the checksum for SS1 again and compare the checksums for the two subsystems again. This time, the checksums should not be equal.

```
[chksm1, chksm1_details] = ...
Simulink.SubSystem.getChecksum(SS1);
isequal(chksm1, chksm2)
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 **chksm1_details**.

```
chksm1_details

chksm1_details =
    ContentsChecksum: [1x1 struct]
    InterfaceChecksum: [1x1 struct]
    ContentsChecksumItems: [221x1 struct]
    InterfaceChecksumItems: [91x1 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 =  
    0
```

In this case, differences exist in both the contents and interface.

- 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  
  
idxForIDiffs=[];
```

```

for idx = 1:length(chksum1_details.InterfaceChecksumItems)
    if (~strcmp(chksum1_details.InterfaceChecksumItems(idx).Identifier, ...
        chksum2_details.InterfaceChecksumItems(idx).Identifier))
        disp(['Identifiers different for interface item ', num2str(idx)]);
        idxForIDiffs=[idxForIDiffs, idx];
    end
    if (ischar(chksum1_details.InterfaceChecksumItems(idx).Value))
        if (~strcmp(chksum1_details.InterfaceChecksumItems(idx).Value, ...
            chksum2_details.InterfaceChecksumItems(idx).Value))
            disp(['String values different for interface item ', num2str(idx)]);
            idxForIDiffs=[idxForIDiffs, idx];
        end
    end
    if (isnumeric(chksum1_details.InterfaceChecksumItems(idx).Value))
        if (chksum1_details.InterfaceChecksumItems(idx).Value ~= ...
            chksum2_details.InterfaceChecksumItems(idx).Value)
            disp(['Numeric values different for interface item ', num2str(idx)]);
            idxForIDiffs=[idxForIDiffs, idx];
        end
    end
end
end

```

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

```

check_details
String values different for contents item 64
String values different for contents item 75
String values different for contents item 81
String values different for interface item 46

```

The results indicate that differences exist for index items 64, 75, and 81 in the subsystem contents and for item 46 in the subsystem interfaces.

- 12** Use the returned index values to get the handle , identifier, and value details for each difference found.

```

chksum1_details.ContentsChecksumItems(64)
ans =
    Handle: 'my_ssreuse/SS1/Lookup Table Output1'
    Identifier: 'CompiledPortAliasedThruDataType'

```

```
        Value: 'int8'
checksum2_details.ContentsChecksumItems(64)
ans =
    Handle: 'my_ssreuse/SS2/Lookup Table Output1'
    Identifier: 'CompiledPortAliasedThruDataType'
        Value: 'double'
checksum1_details.ContentsChecksumItems(75)
ans =
    Handle: 'my_ssreuse/SS1/Lookup Table'
    Identifier: 'RunTimeParameter{'OutputValues'}.DataType'
        Value: 'int8'
checksum2_details.ContentsChecksumItems(75)
ans =
    Handle: 'my_ssreuse/SS2/Lookup Table'
    Identifier: 'RunTimeParameter{'OutputValues'}.DataType'
        Value: 'double'
checksum1_details.ContentsChecksumItems(81)
ans =
    Handle: 'my_ssreuse/SS1/Lookup Table'
    Identifier: 'OutDataTypeMode'
        Value: 'int8'
checksum2_details.ContentsChecksumItems(81)
ans =
    Handle: 'my_ssreuse/SS2/Lookup Table'
    Identifier: 'OutDataTypeMode'
        Value: 'Same as input'
checksum1_details.InterfaceChecksumItems(46)
ans =
    Handle: 'my_ssreuse/SS1'
    Identifier: 'CanonicalParameter(1).DataType'
        Value: 'int8'
checksum2_details.InterfaceChecksumItems(46)
ans =
    Handle: 'my_ssreuse/SS2'
    Identifier: 'CanonicalParameter(1).DataType'
        Value: 'double'
```

As expected, the details identify the Lookup Table block and data type parameters as areas on which to focus for debugging a subsystem reuse issue.

- 13** Correct the problem by changing the output data type mode for the subsystems such that they match.

Supporting Shared Utility Directories in the Build Process

The shared utility directories (`s1prj/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 directory for a standalone model. See Chapter 39, “Setting Up Runtime Logging to MAT-Files” for details.

If you want your target to support compilation of code generated in the shared utilities directory, several updates to your template makefile (TMF) are required. Support for the shared utilities directory is a necessary, but not sufficient, condition for supporting model reference builds. See “Supporting Model Referencing” on page 21-110 to learn about additional updates that are needed 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 directories:

- 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.

Modifying Template Makefiles to Support Shared Utilities

Make the following changes to your TMF to support the shared utilities directory:

- 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 directory 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 directories for shared source files and the library compiled from the source files. In the current release, all 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 =
ifeq ($SHARED_SRC_DIR, )
SHARED_INCLUDES = -I$(SHARED_SRC_DIR)
endif

INCLUDES = -I. $(MATLAB_INCLUDES) $(ADD_INCLUDES) \
           $(USER_INCLUDES) $(SHARED_INCLUDES)
```

- 3** Update the SHARED_SRC variable to list all 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 any 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)
```

Creating Component Object Libraries and Enhancing Simulation Performance

S-functions are an important class of target for which the Real-Time Workshop 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 demonstrate how to generate them. For more details on the structure of S-functions, see the Simulink Writing S-Functions documentation.

In this section...

- “Introduction” on page 3-92
- “Creating an S-Function Block from a Subsystem” on page 3-95
- “Tunable Parameters in Generated S-Functions” on page 3-101
- “Automated S-Function Generation” on page 3-104
- “System Target File and Template Makefiles” on page 3-108
- “Checksums and the S-Function Target” on page 3-109
- “S-Function Target Limitations” on page 3-109

Introduction

- “S-Function Target Overview” on page 3-92
- “Required Files for S-Function Deployment” on page 3-94
- “Sample Time Propagation in Generated S-Functions” on page 3-95
- “Choice of Solver Type” on page 3-95

S-Function Target Overview

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 “Creating an S-Function Block from a Subsystem” on page 3-95 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 or real-time malloc targets. 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 format. This allows any level of nested S-functions.

You should avoid nesting S-functions 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 Real-Time Workshop 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, you should be sure to name the subsystem or model to be generated as an S-function target uniquely, to avoid duplicating any existing MEX filenames on the MATLAB path.

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 from which all block and line information has been eliminated using the Model Protection facility. For more information, see “Protecting Referenced Models” in the Simulink documentation.
 - The Real-Time Workshop 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 “Creating and Using Host-Based Shared Libraries” in the Real-Time Workshop 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 directory 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 directory 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*)

- Subdirectory `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 “Inheriting Sample Times” in the Simulink documentation and “Inherited Sample Time for Referenced Models” on page 3-66 in the Real-Time Workshop documentation.

To ensure that your generated S-Function block 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.

Choice of Solver Type

If the model containing the subsystem from which you generate an S-function uses a variable-step solver, the generated S-function contains zero-crossing functions and will work properly only in models that use variable-step solvers.

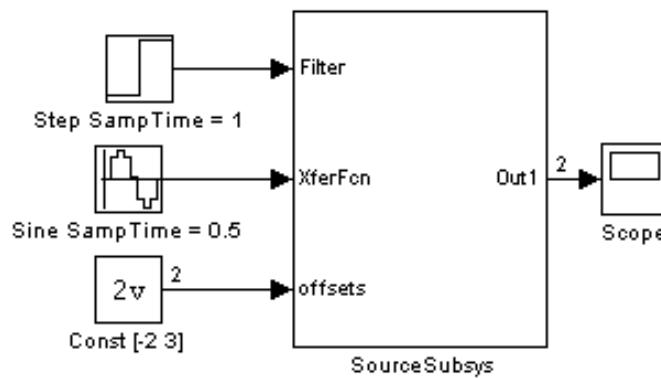
If the model containing the subsystem from which you generate an S-function uses a fixed-step solver, the generated S-function contains no zero-crossing functions and the generated S-function will work properly in models that use variable-step or fixed-step solvers.

Creating an S-Function Block from a Subsystem

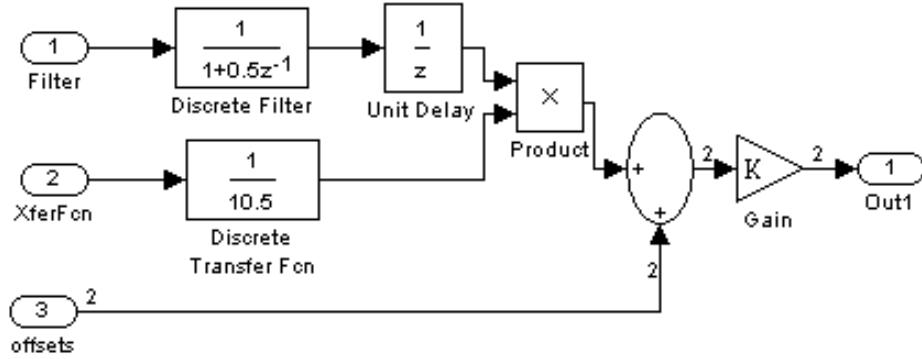
This section demonstrates how to extract a subsystem from a model and generate a reusable S-function component from it.

The next figure shows `SourceModel1`, 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

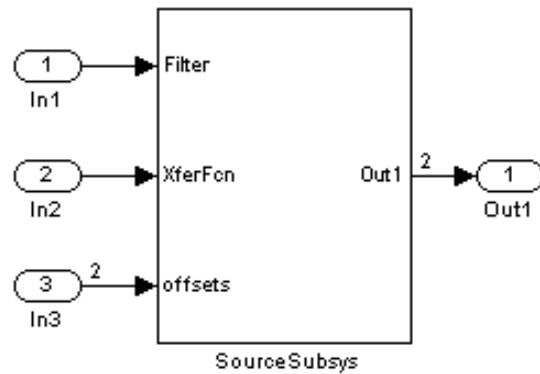
The objective is to extract **SourceSubsys** from the model and build an S-Function block from it, using the S-function target. The S-Function block must perform identically to the subsystem from which it was generated.

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 3-95.)

In this example, you want the S-Function block to retain the properties of **SourceSubsys** as it exists in **SourceModel1**. Therefore, before you build the subsystem as a separate S-function component, you must set the import 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. This ensures that the generated S-function component will operate identically to the original subsystem (see “Choice of Solver Type” on page 3-95 for an exception to this rule).

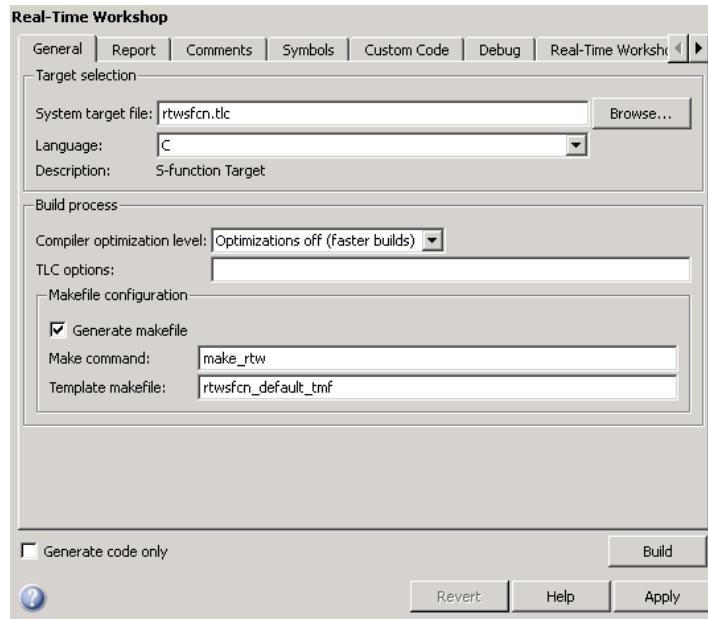
To build **SourceSubsys** as an S-function component,

- 1** Create a new model and copy/paste **SourceSubsys** into the empty window.
- 2** Set the signal widths and sample times of imports inside **SourceSubsys** such that they match those of the signals in the original model. Import 1, **Filter**, has a width of 1 and a sample time of 1. Import 2, **Xferfcn**, has a width of 1 and a sample time of 0.5. Import 3, **offsets**, has a width of 2 and a sample time of 0.5.
- 3** The generated S-Function block should have three imports and one outport. Connect imports and an outport to **SourceSubsys**, as shown in the next figure.

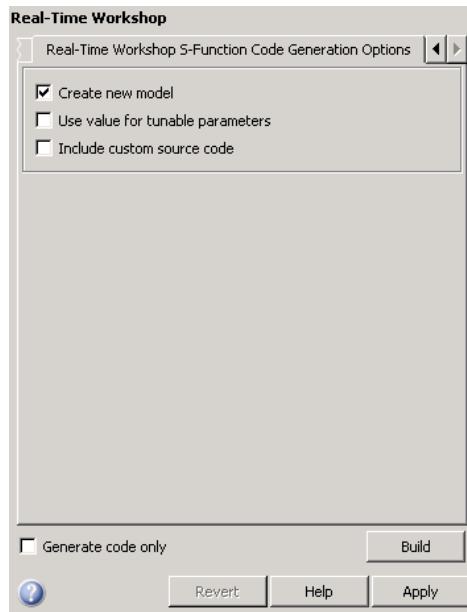


The correct signal widths and sample times are propagated to these ports.

- 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 Model Explorer or the Configuration Parameters dialog box, click the **Real-Time Workshop** tab.
- 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 **Real-Time Workshop** pane appears as follows.



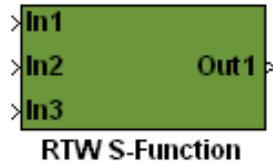
- 8 Select the **Real-Time Workshop S-Function Code Generation Options** tab (in Model Explorer) or pane (in the Configuration Parameters dialog box).
- 9 Make sure that **Create New Model** is selected, as shown in the next figure:



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** if necessary.

- 10 Save the new model containing your subsystem, for example as `SourceSubsys.mdl`.
- 11 Build the model.
- 12 The Real-Time Workshop build process builds the S-function component in the working directory. After the build, a new model window is displayed.

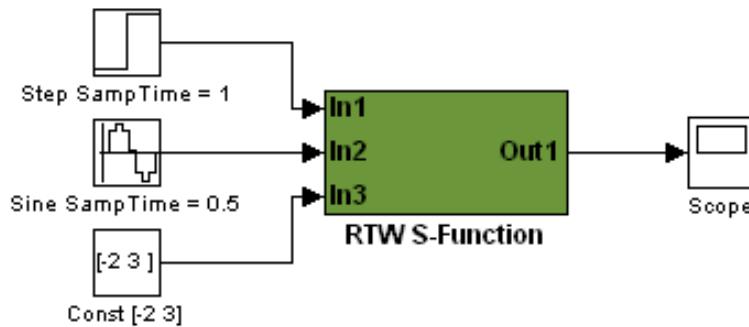


Optionally you can save the generated model, for example as `SourceSubsys_Sfunction.mdl`.

- 13** You can now copy the Real-Time Workshop 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 3-94.

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:

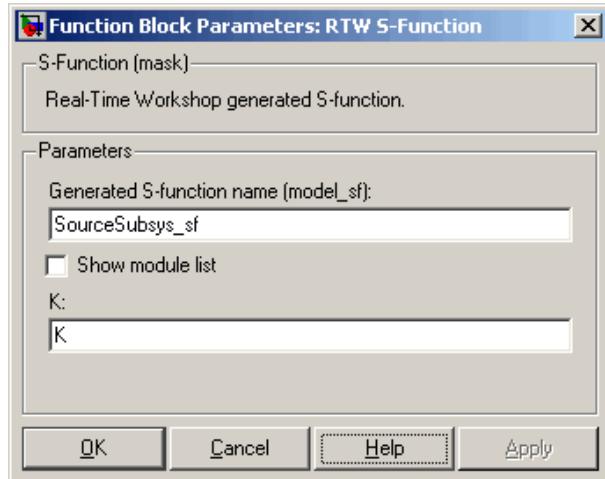
- Use the **Generate S-function** feature (see “Automated S-Function Generation” on page 3-104).
- or
- Use the Model Parameter Configuration dialog box (see Chapter 10, “Parameter Considerations”) to declare desired block parameters tunable.

Block parameters that are declared tunable with the `auto` storage class in the source model become tunable parameters of the generated S-function.

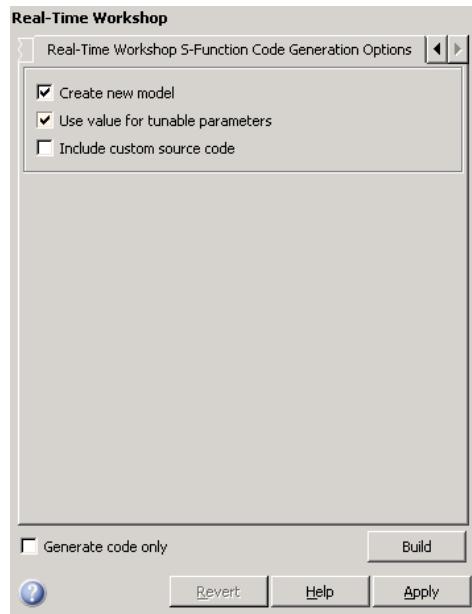
These parameters do not become part of a generated `model_P` (formerly `rtP`) parameter data structure, as they would in code generated from other targets. Instead, the generated code accesses these parameters by using MEX API calls such as `mxGetPr` or `mxGetData`. Your code should access these parameters in the same way.

For more information on MEX API calls, see “Writing S-Functions in C” in the Simulink Writing S-Functions documentation and External Interfaces in the MATLAB online documentation.

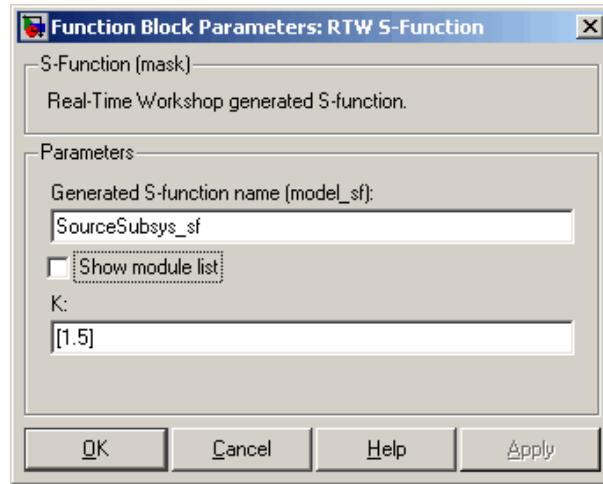
S-Function blocks created by using the S-function target are automatically masked. The mask displays each tunable parameter in an edit field. By default, the edit field displays the parameter by variable name, as in the following example.



You can choose to display the value of the parameter rather than its variable name by selecting **Use Value for Tunable Parameters** in the **Options** section.



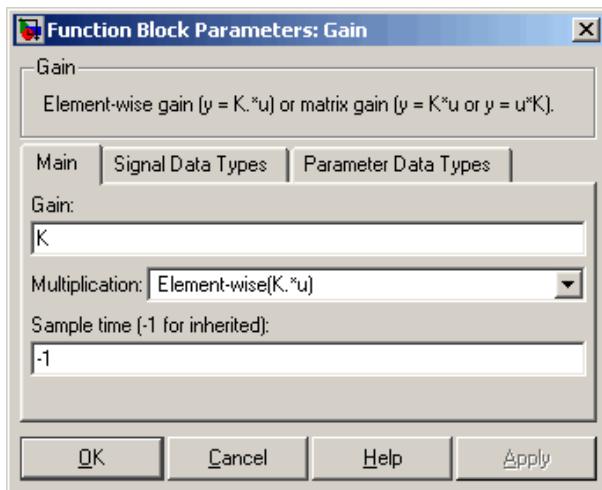
When this option is chosen, the value of the variable (at code generation time) is displayed in the edit field, as in the following example.



Automated 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 previous example, “Creating an S-Function Block from a Subsystem” on page 3-95. 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 gain factor of the Gain block should be set within **SourceSubsys** to the workspace variable **K** (as illustrated in the next figure) and declare **K** as a tunable parameter.



To auto-generate an S-function from **SourceSubsys** with tunable parameter **K**,

- 1** Click the **SourceSubsys** subsystem to select it.
- 2** Select **Generate S-function** from the **Real-Time Workshop** submenu of the **Tools** menu. This menu item is enabled when a subsystem is selected in the current model.

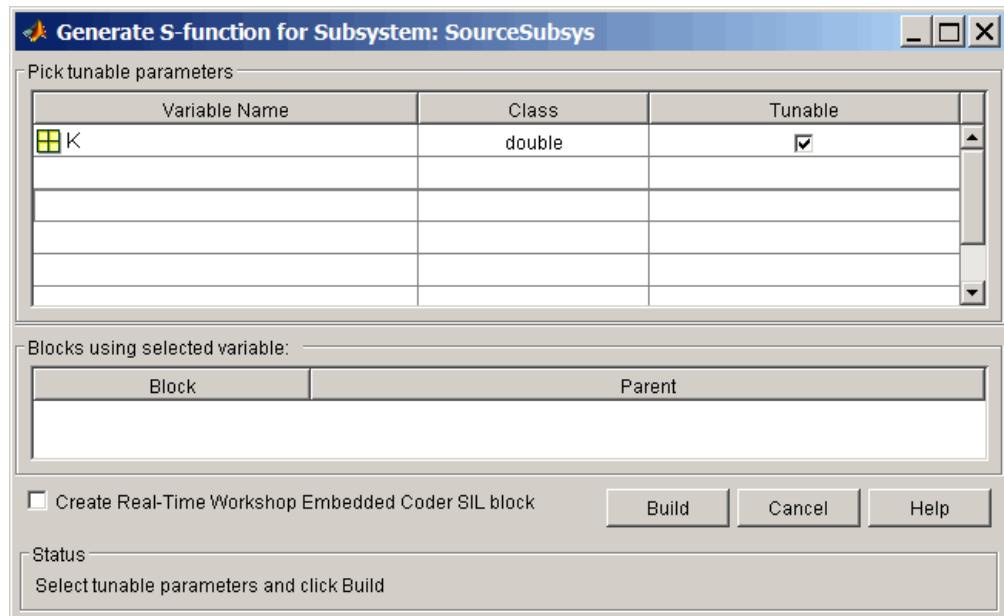
Alternatively, you can right-click the subsystem and select **Real-Time Workshop > Generate S-Function** from the subsystem block’s context menu.

- 3** The **Generate S-function** window is displayed (see the next figure). This window shows all 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 all the blocks that reference the parameter, and the parent system of each such block.

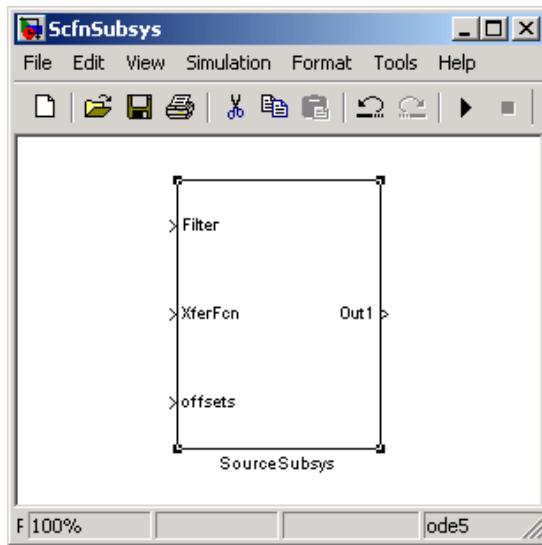


Generate S-Function Window

- 4 If you have installed the Real-Time Workshop Embedded Coder product, and if the subsystem does not have a continuous sample time, the **Create Real-Time Workshop Embedded Coder SIL block** check box is

available, as shown above. Otherwise, it appears dimmed. When **Create Real-Time Workshop Embedded Coder SIL block** is selected, the build process generates a wrapper S-function by using the Real-Time Workshop Embedded Coder product. See “Generating S-Function Wrappers” in the Real-Time Workshop Embedded Coder documentation for more information.

- 5 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.
- 6 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.



- 7 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.
- 8 The generated code for the S-Function block is stored in the current working directory. The following files are written to the top level directory:
 - *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 subdirectory *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 3-94.

- 9 The generated S-Function block has imports 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 always set to its default value (off).
 - A MEX S-function wrapper must only be used in the MATLAB version in which the wrapper is created.
-

System Target File and Template Makefiles

- “Introduction” on page 3-108
- “System Target File” on page 3-108
- “Template Makefiles” on page 3-108

Introduction

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 Real-Time Workshop 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 Real-Time Workshop 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 Real-Time Workshop software applies the subsystem and model checksums as follows:

- Code reuse — If two subsystems in a model have the same checksum, the Real-Time Workshop build process generates code for one function only.
- Model reference — If the current model checksum matches the checksum when the model was built, the Real-Time Workshop build process does not rebuild submodels.
- External mode — If the current model checksum does not match the checksum of the code that is running on the target, the Real-Time Workshop build process generates an error.

S-Function Target Limitations

- “Limitations on Using Tunable Variables in Expressions” on page 3-110
- “Run-Time Parameters and S-Function Compatibility Diagnostics” on page 3-110
- “Limitations on Using Goto and From Block” on page 3-110
- “Limitations on Building and Updating S-Functions” on page 3-112

- “Unsupported Blocks” on page 3-112
- “SimState Not Supported for Code Generation” on page 3-113
- “Profiling Code Performance Not Supported” on page 3-113

Limitations on Using Tunable Variables in Expressions

Certain limitations apply to the use of tunable variables in expressions. When Real-Time Workshop 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 10-18.

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 Real-Time Workshop software reports that an upgrade is needed for any S-function 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**. Alternatively, if you have a Real-Time Workshop Embedded Coder license, select the **Create Real-Time Workshop Embedded Coder SIL block** check box on the **Generate S-function for Subsystem** dialog box and create a SIL block (containing the ERT S-function). In this case, you do not receive the upgrade messages. However, you cannot include SIL blocks inside other generated S-functions recursively.

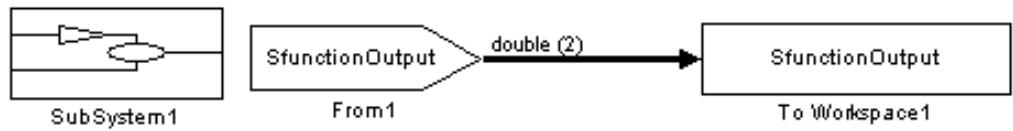
Limitations on Using Goto and From Block

When using the S-function target, the Real-Time Workshop code generator restricts I/O to correspond to the root model’s Import and Outport blocks (or the Import and Outport blocks of the Subsystem block from which the S-function target was generated). No code is generated for Goto or From blocks.

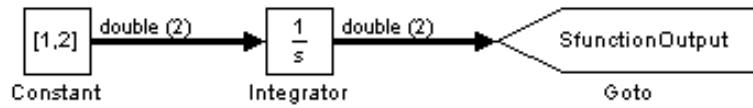
To work around this restriction, create your model and subsystem with the required Import and Outport blocks, instead of using Goto and From blocks to pass data between the root model and subsystem. In the model that incorporates the generated S-function, you would then add needed Goto and From blocks.

Example Before Work Around

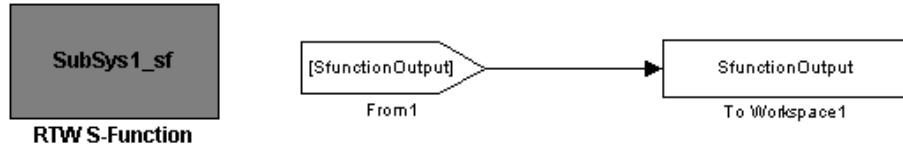
- Root model with a From block and subsystem, Subsystem1



- Subsystem1 with a Goto block, which has global visibility and passes its input to the From block in the root model

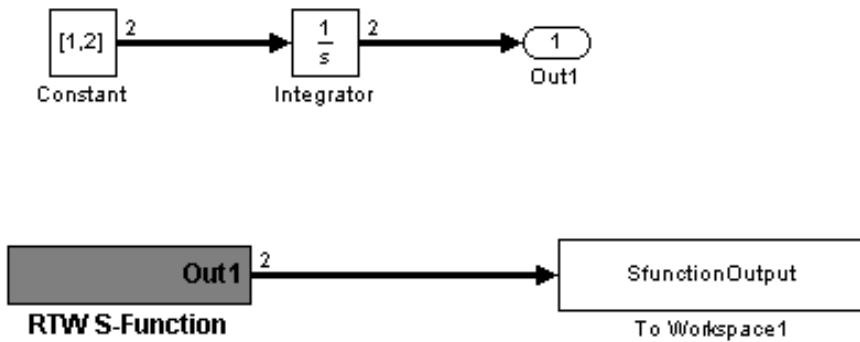


- Subsystem1 replaced with an S-function generated with the S-Function target — a warning results when you run the model because the generated S-function does not implement the Goto block



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 Real-Time Workshop S-function target:

- You cannot build models that contain Model blocks using the Real-Time Workshop S-function target. This also means that you cannot build a subsystem module by right-clicking (or by using **Tools > Real-Time Workshop > Build 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 Real-Time Workshop 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 options** 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 Writing S-Functions documentation.

Unsupported Blocks

The S-function format does not support the following built-in blocks:

- MATLAB Fcn 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

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 Real-Time Workshop S-function target.

Profiling Code Performance Not Supported

Profiling the performance of generated code using the Target Language Compiler (TLC) hook function interface described in “Profiling Code Performance” on page 43-7 is not supported for the S-function target.

Combining Multiple Models

If you want to combine several models (or several instances of the same model) into a single executable, the Real-Time Workshop 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 “Referencing a Model” and “Creating Model Components” on page 3-51.

When developing embedded systems using the Real-Time Workshop 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, Real-Time Workshop Embedded Coder target has certain restrictions that might not be appropriate for your application. For more information, see the Real-Time Workshop Embedded Coder documentation.

The GRT malloc target is another possible solution. Using it is appropriate in situations where you want do any or all of the following:

- Selectively control calls to more than one model
- Use dynamic memory allocation
- Include models that employ continuous states
- Log data to multiple files
- Run one of the models in external mode

To summarize by target, your options are as follows:

Target	Support for Combining Multiple Models?
Generic Real-Time Target (<code>grt.tlc</code>)	Yes (using Model blocks)
Generic Real-Time Target with dynamic memory allocation (<code>grt_malloc.tlc</code>)	Yes

Target	Support for Combining Multiple Models?
Real-Time Workshop Embedded Coder (<code>ert.tlc</code>)	Yes
S-function Target (<code>rtwsfcn.tlc</code>)	No

Using GRT Malloc to Combine Models

This section discusses how to use the GRT malloc target 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.
- 2 Combine the makefiles for each of the models into one makefile for creating the final multimodel executable.
- 3 Create a combined simulation engine by modifying `grt_malloc_main.c` to initialize and call the models correctly.
- 4 Run the combination makefile to link the object files from the models and the main program into an executable.

Sharing Data Across Models

It is safest to 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 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 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 `grt_malloc_main`) must set up the timer interrupt to occur at the greatest common divisor rate of the models. The main program is responsible for calling each of the models at the appropriate 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.

Block Support Considerations

- “Simulink Built-In Blocks That Support Code Generation” on page 4-2
- “Block Set Support for Code Generation” on page 4-23
- “User-Created Block Considerations” on page 4-24

Simulink Built-In Blocks That Support Code Generation

The following tables summarize Real-Time Workshop and Real-Time Workshop Embedded Coder support for Simulink blocks. There is a table for each block library. For each block, the second column indicates any support notes, which give information about the block for code generation. 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 on page 4-3
- Additional Math and Discrete: Increment/Decrement on page 4-4
- Continuous on page 4-4
- Discontinuities on page 4-5
- Discrete on page 4-6
- Logic and Bit Operations on page 4-8
- Lookup Tables on page 4-9
- Math Operations on page 4-10
- Model Verification on page 4-13
- Model-Wide Utilities on page 4-14
- Ports & Subsystems on page 4-14
- Signal Attributes on page 4-15
- Signal Routing on page 4-16
- Sinks on page 4-17
- Sources on page 4-19
- User-Defined on page 4-22

Additional Math and Discrete: Additional Discrete

Block	Support Notes
Fixed-Point State-Space	<p>The Real-Time Workshop 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.</p>
Transfer Fcn Direct Form II	
Transfer Fcn Direct Form II Time Varying	
Unit Delay Enabled	
Unit Delay Enabled External IC	
Unit Delay Enabled Resettable	
Unit Delay Enabled Resettable External IC	
Unit Delay External IC	<ul style="list-style-type: none"> The Real-Time Workshop 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.
Unit Delay Resettable	<ul style="list-style-type: none"> Generated code might rely on <code>memcpy</code> or <code>memset(string.h)</code>.
Unit Delay Resettable External IC	
Unit Delay With Preview Enabled	
Unit Delay With Preview Enabled Resettable	
Unit Delay With Preview Enabled Resettable External RV	
Unit Delay With Preview Resettable	
Unit Delay With Preview Resettable External RV	

Additional Math and Discrete: Increment/Decrement

Block	Support Notes
Decrement Real World	The Real-Time Workshop 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	
Increment Real World	
Increment Stored Integer	The Real-Time Workshop 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.

Continuous

Block	Support Notes
Derivative	
Integrator	
Integrator Limited	
PID Controller	
PID Controller (2DOF)	
Second-Order Integrator	
Second-Order Integrator Limited	
State-Space	
Transfer Fcn	
Transport Delay	
Variable Time Delay	<ul style="list-style-type: none"> • Consider using the Simulink Model Discretizer to map continuous blocks (except for the Second-Order Integrator blocks) into discrete equivalents that support code generation. To start the Model Discretizer, select Tools > Control Design > Model Discretizer. • 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 correct 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.

Continuous (Continued)

Block	Support Notes
Variable Transport Delay	
Zero-Pole	

Discontinuities

Block	Support Notes
Backlash	Supports code generation.
Coulomb and Viscous Friction	The Real-Time Workshop 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 Real-Time Workshop 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 correct 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.

Discontinuities (Continued)

Block	Support Notes
Rate Limiter Dynamic	The Real-Time Workshop 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	
Saturation Dynamic	
Wrap To Zero	The Real-Time Workshop 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.

Discrete

Block	Support Notes
Difference	<ul style="list-style-type: none"> • The Real-Time Workshop 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 correct 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 (Continued)

Block	Support Notes
Discrete Derivative	<ul style="list-style-type: none"> Generated code might rely on <code>memcpy</code> or <code>memset</code> (<code>string.h</code>). Depends on absolute time when used inside a triggered subsystem hierarchy.
Discrete Filter	Support code generation.
Discrete FIR Filter	
PID Controller	<ul style="list-style-type: none"> Generated code might rely on <code>memcpy</code> or <code>memset</code> (<code>string.h</code>). Depends on absolute time when used inside a triggered subsystem hierarchy.
PID Controller (2DOF)	
Discrete State-Space	Generated code might rely on <code>memcpy</code> or <code>memset</code> (<code>string.h</code>).
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 correct 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.
Integer Delay	Support code generation.
Memory	
Tapped Delay	

Discrete (Continued)

Block	Support Notes
Transfer Fcn First Order	
Transfer Fcn Lead or Lag	
Transfer Fcn Real Zero	The Real-Time Workshop 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.
Unit Delay	Generated code might rely on <code>memcpy</code> or <code>memset</code> (<code>string.h</code>).
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	
Detect Change	Generated code might rely on <code>memcpy</code> or <code>memset</code> (<code>string.h</code>).
Detect Decrease	
Detect Fall Negative	
Detect Fall Nonpositive	
Detect Increase	
Detect Rise Nonnegative	
Detect Rise Positive	

Logic and Bit Operations (Continued)

Block	Support Notes
Extract Bits	Support code generation.
Interval Test	
Interval Test Dynamic	
Logical Operator	
Relational Operator	
Shift Arithmetic	

Lookup Tables

Block	Support Notes
Cosine	The Real-Time Workshop 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.
Direct Lookup Table (n-D)	Support code generation.
Interpolation Using Prelookup	
Lookup Table	
Lookup Table (2-D)	
Lookup Table (n-D)	
Lookup Table Dynamic	
Prelookup	
Sine	The Real-Time Workshop 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

Lookup Tables (Continued)

Block	Support Notes
	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)	

Math Operations (Continued)

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	

Math Operations (Continued)

Block	Support Notes
Sine Wave Function	<ul style="list-style-type: none">• Does not refer to absolute time when configured for sample-based operation. Depends on absolute time when in time-based operation.• Depends on absolute time when used inside a triggered subsystem hierarchy.
Slider Gain	Support code generation.
Sqrt	
Squeeze	
Subtract	
Sum	
Sum of Elements	
Trigonometric Function	Functions <code>asinh</code> , <code>acosh</code> , and <code>atanh</code> are not supported by all compilers. If you use a compiler that does not support the functions, the Real-Time Workshop software issues a warning message 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 correct 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 memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally correct and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.
Check Static Gap	
Check Static Lower Bound	
Check Static Range	
Check Static Upper Bound	

Model-Wide Utilities

Block	Support Notes
Block Support Table	Ignored during code generation.
DocBlock	Uses the template symbol you specify for the RTW Embedded Coder Flag block parameter to add comments to generated code. Requires a Real-Time Workshop Embedded Coder license. For more information, see “Using 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	
Function-Call Split	
Function-Call Subsystem	
If	

Ports & Subsystems (Continued)

Block	Support Notes
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 correct 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.

Signal Attributes (Continued)

Block	Support Notes
Rate Transition	<ul style="list-style-type: none">Generated code might rely on <code>memcpy</code> or <code>memset</code> (<code>string.h</code>).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 correct and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

Signal Routing (Continued)

Block	Support Notes
From	Support code generation.
Goto	
Goto Tag Visibility	
Index Vector	
Manual Switch	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally correct 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 <i>be identically labeled and have the same storage class</i> . When Merge blocks connect directly to one another, these rules apply to all signals connected to Merge blocks in the group.
Multiport Switch	Support code generation.
Mux	
Selector	
Switch	Generated code might rely on <code>memcpy</code> or <code>memset (string.h)</code> .

Sinks

Block	Support Notes
Display	Ignored for code generation.
Floating Scope	
Outport (Out1)	Supports code generation.

Sinks (Continued)

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

Sources (Continued)

Block	Support Notes
	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 correct 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.
Import (In1)	
Pulse Generator	Cannot use inside a triggered subsystem hierarchy. Does not refer to absolute time when configured for sample-based operation. Depends on absolute time when in time-based operation.
Ramp	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally correct 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.

Sources (Continued)

Block	Support Notes
Repeating Sequence	<ul style="list-style-type: none"> Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally correct and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable. Consider using the Repeating Sequence Stair or Repeating Sequence Interpolated block instead.
Repeating Sequence Interpolated	<ul style="list-style-type: none"> The Real-Time Workshop 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	<p>The Real-Time Workshop 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.</p>
Signal Builder	<p>Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally correct and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.</p>
Signal Generator	<p>Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally correct and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.</p>

Sources (Continued)

Block	Support Notes
Sine Wave	<ul style="list-style-type: none">Depends on absolute time when used inside a triggered subsystem hierarchy.Does not refer to absolute time when configured for sample-based operation. Depends on absolute time when in time-based operation.
Step	Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally correct 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
Embedded MATLAB Function	Support code generation.
Fcn	
Level-2 M-File S-Function	Ignored during code generation.
MATLAB Fcn	Consider using the Embedded MATLAB Function block instead.
S-Function	
S-Function Builder	S-functions that call into MATLAB are not supported for code generation. See “User-Created Block Considerations” on page 4-24 for advice on generating optimized user-defined blocks.

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 whether any, all, or a subset of blocks support code generation.

User-Created Block Considerations

- If possible, use one of the following mechanisms to automate block creation:
 - Legacy Code Tool (see “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation)
 - S-Function Builder block (see the block description)
- If you create a block from an S-function, rather than as a masked subsystem or built-in Simulink blocks, inline the S-function, using a Target Language Compiler (TLC) file. The easiest way to inline external code is to use the Legacy Code Tool. For more information, see “Inlining S-Functions with TLC” in the Target Language Compiler documentation and the following Simulink topics:
 - “Creating Custom Blocks”
 - “Selecting an S-Function Implementation”
 - “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool”
- If you create a block for a model for which generated code is intended for production deployment, avoid:
 - Accessing absolute time values. If a block depends on absolute time and the time value 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 no longer correct. For more information, see “Using Timers” on page 5-70.
 - Avoid using continuous sample times. Although you can generate code for blocks that use continuous time (for example, if you specify a fixed-step ODE solver), the execution performance and stability of code generated from models that do not use continuous sample times is much better. For more information, see “Using Discrete and Continuous Time”.
 - If possible, avoid creating multirate S-function blocks. Such blocks have complex data handling requirements when you use them in a multitasking solver mode. For more information, see “Creating Multitasking-Safe, Multirate, Port-Based Sample Time S-Functions” on page 32-83.

Scheduling Considerations

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

- “Introduction” on page 5-2
- “Single-Tasking and Multitasking Execution Modes” on page 5-3
- “Handling Rate Transitions” on page 5-13
- “Example: Single-Tasking and Multitasking Execution of a Model” on page 5-27
- “Handling Asynchronous Events” on page 5-34
- “Using Timers” on page 5-70
- “Configuring Scheduling” on page 5-83

Introduction

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 correct real-time code for execution in a multitasking environment. To make multirate models operate correctly 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 successfully in a multitasking environment. For a comprehensive discussion of sample times, including rate transitions, see “Working with Sample Times” in the Simulink User’s Guide.

Single-Tasking and Multitasking Execution Modes

In this section...

- “Introduction” on page 5-3
- “Executing Multitasking Models” on page 5-5
- “Multitasking and Pseudomultitasking Modes” on page 5-6
- “Building a Program for Multitasking Execution” on page 5-9
- “Single-Tasking Mode” on page 5-9
- “Building a Program for Single-Tasking Execution” on page 5-10
- “Model Execution and Rate Transitions” on page 5-10
- “Simulating Models with the Simulink Product” on page 5-11
- “Executing Models in Real Time” on page 5-11
- “Single-Tasking Versus Multitasking Operation” on page 5-12

Introduction

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.

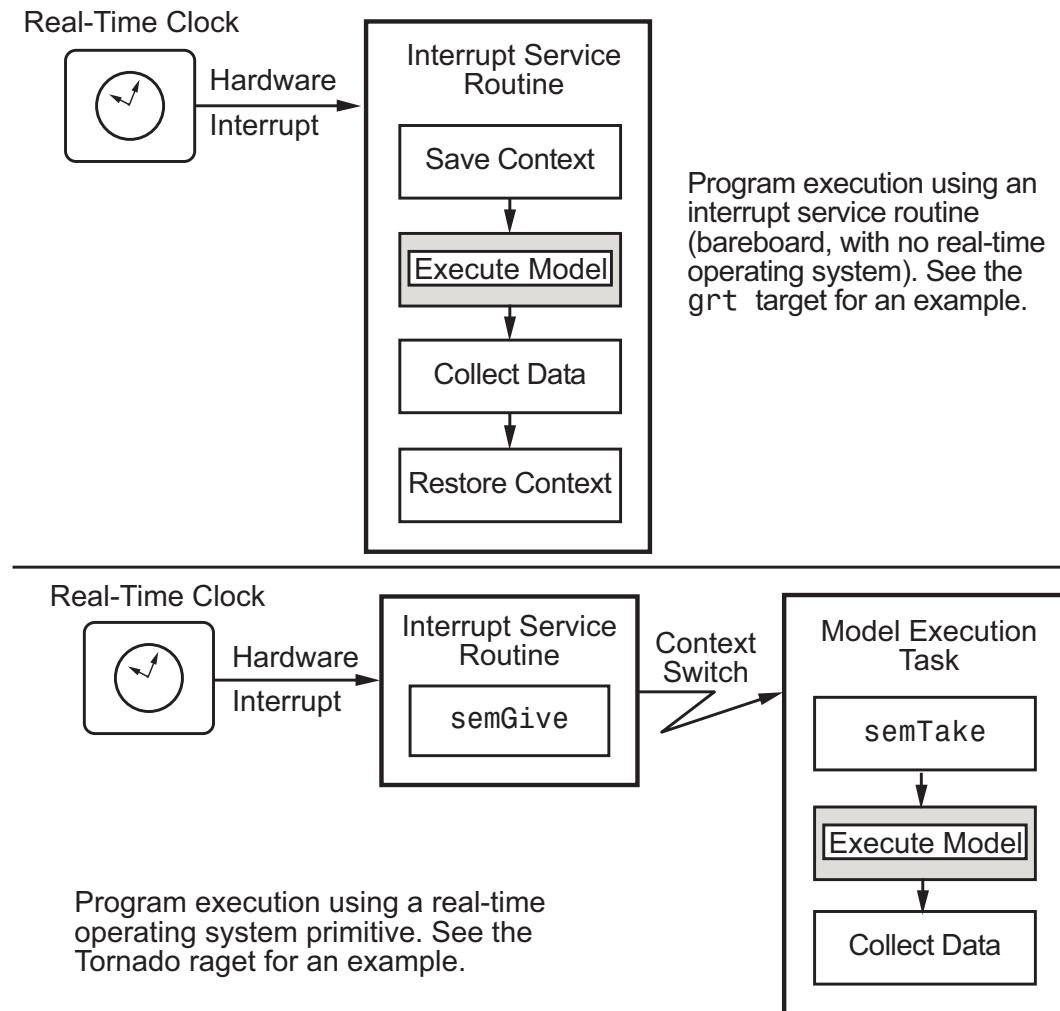
Execution of models in a real-time system can be done with the aid of a real-time operating system, or it can be done on a *bare-board* target, where the model runs in the context of an interrupt service routine (ISR).

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

In operating systems (such as PC-DOS) where only one process can exist at any 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.

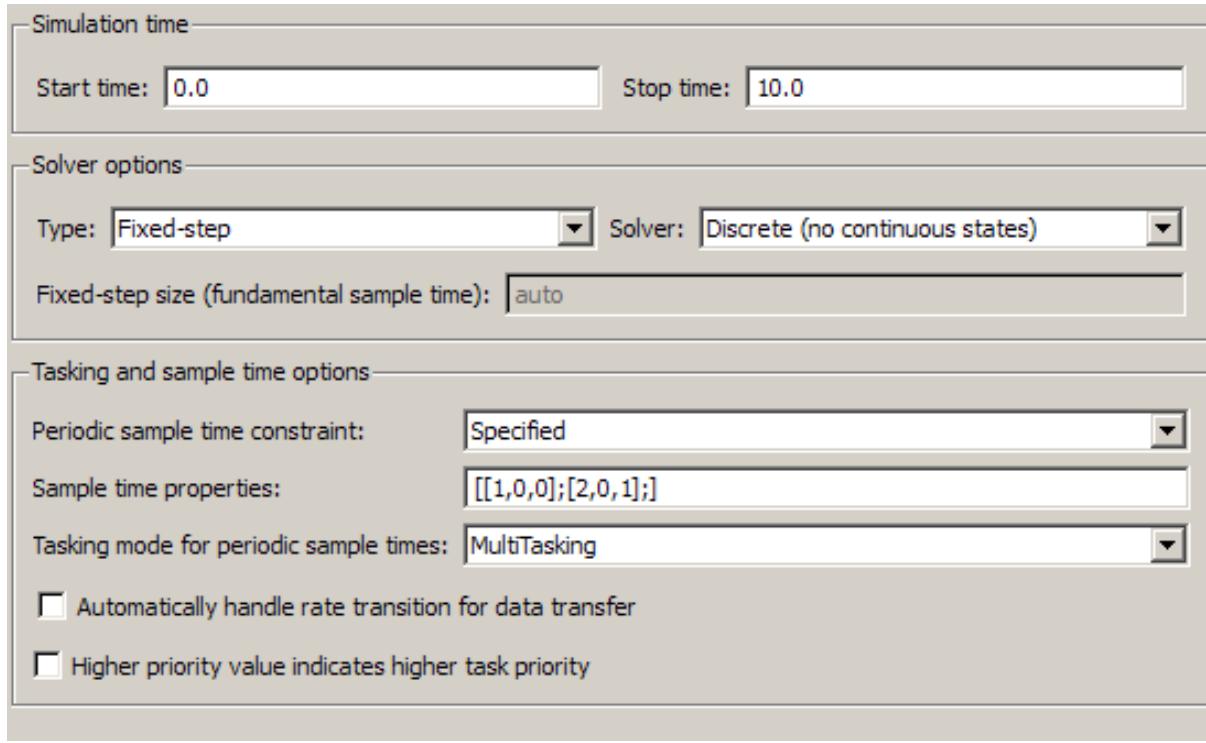


Executing Multitasking Models

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 Real-Time Workshop 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 any 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 demo model `rtwdemo_mrmtbb` shows an example.



- Continuous blocks always 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.

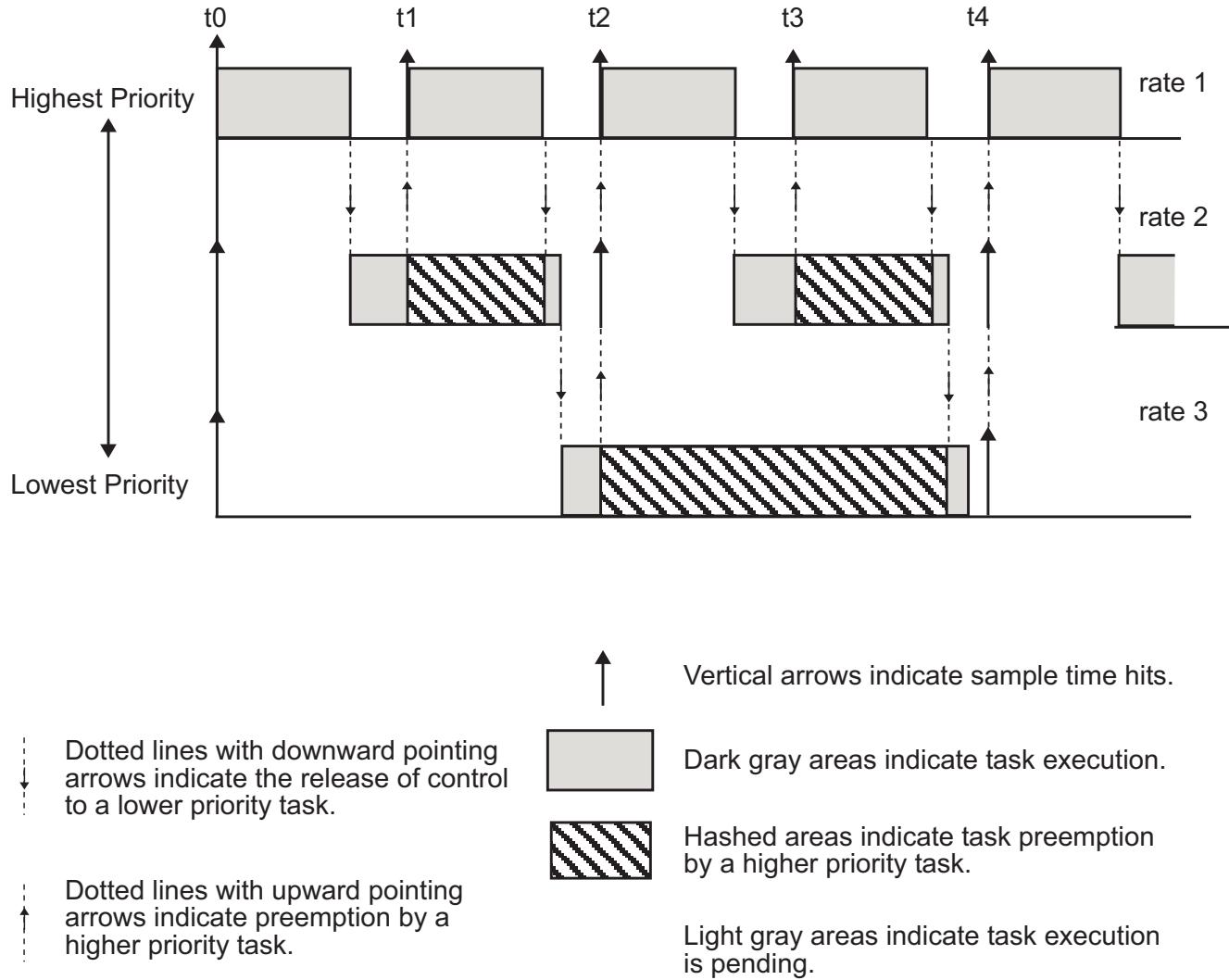
Where tasks are asynchronous rather than periodic, there may not necessarily be a relationship between sample rates and task priorities; the task with the highest priority need not have the fastest sample rate. You specify asynchronous task priorities using Async Interrupt and Task Synchronization blocks.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

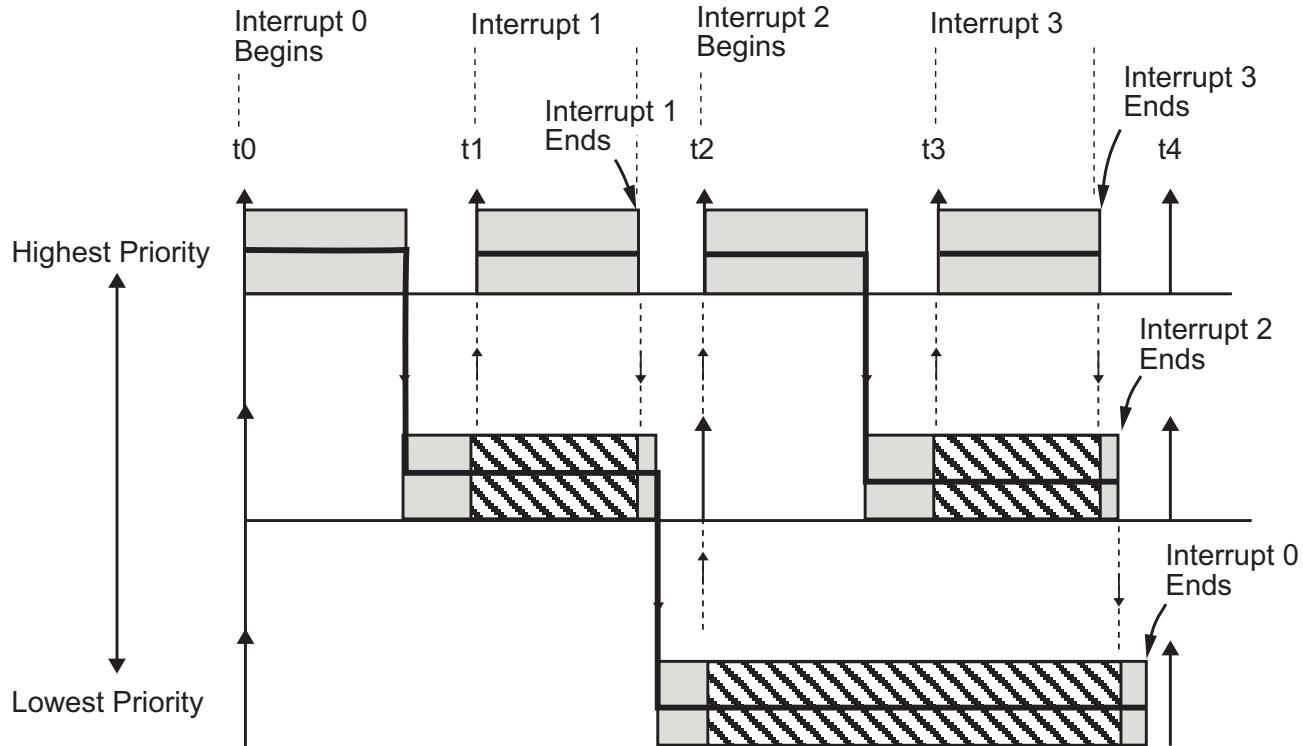
In multitasking environments (that is, under a real-time operating system), you can define separate tasks and assign them priorities. In a bare-board target (that is, no real-time operating system present), you cannot create separate tasks. However, Real-Time Workshop application modules implement what 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 Real-Time Workshop software in multitasking, pseudomultitasking, and single-tasking environments.

5 Scheduling Considerations





Building a Program for Multitasking Execution

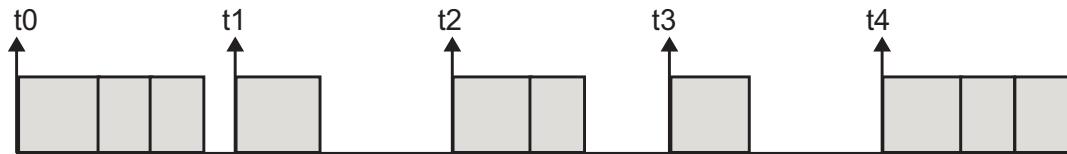
To use multitasking execution, select Auto (the default) or MultiTasking from the **Tasking mode for periodic sample times** menu on the **Solver** pane of the Configuration Parameters dialog box. This menu is active only if you select **Fixed-step** as the solver type. Auto mode results in a multitasking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

Building 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 correctly in real time, you (or the Simulink engine) might need to identify and properly 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 5-18 for an explanation of this option.

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

Simulating Models with the Simulink Product

Before the Simulink engine simulates a model, it orders all 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.

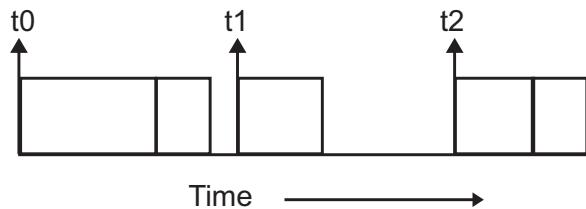
The key to this process is the proper ordering of blocks. Any block whose output is directly dependent on its input (that is, any block with direct feedthrough) cannot execute until the block driving its input executes.

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, all necessary computations are performed prior to advancing the variable corresponding to time. In essence, this results in all computations occurring instantaneously (that is, no computational delay).

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

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

See “Multitasking and Pseudomultitasking Modes” on page 5-6 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 Simulink model by adding Rate Transition blocks (or instruct the Simulink engine to do so) to generate correct results. The next section, “Handling Rate Transitions” on page 5-13, discusses issues related to rate transition blocks.

Handling Rate Transitions

In this section...

- “Introduction” on page 5-13
- “Data Transfer Problems” on page 5-15
- “Data Transfer Assumptions” on page 5-16
- “Rate Transition Block Options” on page 5-16
- “Faster to Slower Transitions in a Simulink Model” on page 5-21
- “Faster to Slower Transitions in Real Time” on page 5-21
- “Slower to Faster Transitions in a Simulink Model” on page 5-23
- “Slower to Faster Transitions in Real Time” on page 5-24

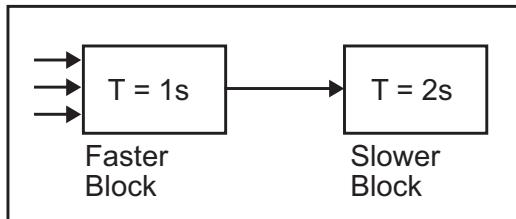
Introduction

Two periodic sample rate transitions can exist within a model:

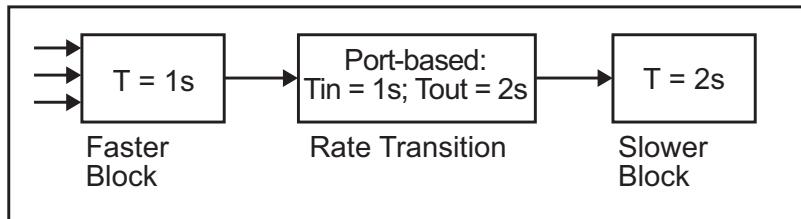
- A faster block driving a slower block
- A slower block driving a faster block

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 “Handling Asynchronous Events” on page 5-34.

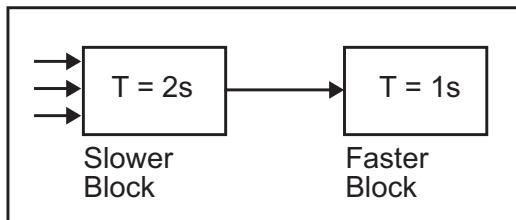
In single-tasking systems, there are no issues involving multiple sample rates. In multitasking and pseudomultitasking systems, however, differing sample rates can cause problems by causing blocks to be executed in the wrong order. To prevent possible errors in calculated data, you must control model execution at these transitions. When connecting faster and slower blocks, you or the Simulink engine must add Rate Transition blocks between them. Fast-to-slow transitions are illustrated in the next figure.



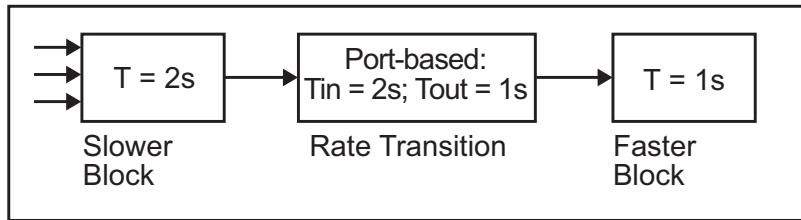
becomes



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



becomes



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

Data Transfer Problems

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

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

Consider the following scenario:

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

Such a data transfer is called *unprotected*. “Faster to Slower Transitions in Real Time” on page 5-21 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 ensure that data transfers in your application are both protected and 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 Real-Time Workshop software assumes the following:

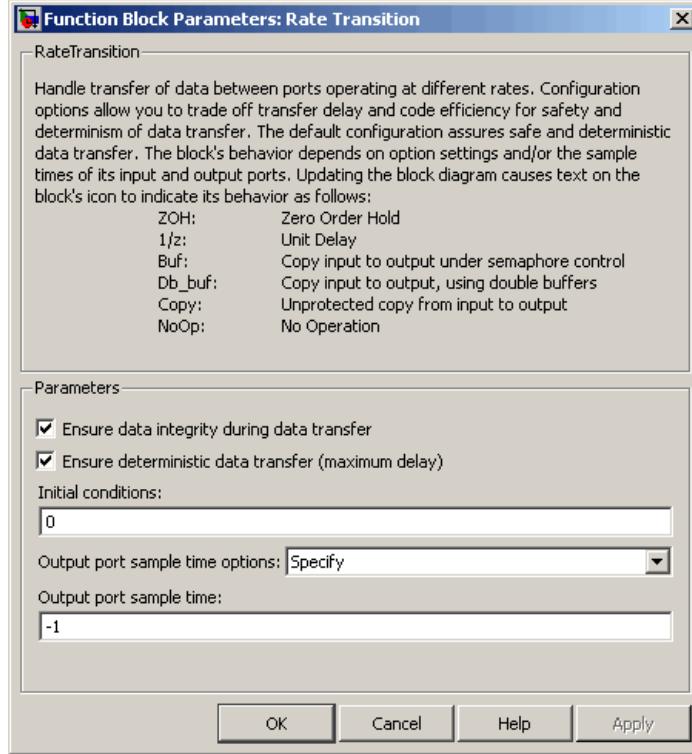
- Data transitions occur between a single reading task and a single writing task.
- A read or write of a byte-sized variable is atomic.
- When two tasks interact through a data transition, only one of them can preempt the other.
- For periodic tasks, the faster rate task has higher priority than the slower rate task; the faster rate task always 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 appropriate 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, the integrity of data transferred between rates is guaranteed (the data transfer is protected). When this option is off, data integrity is not guaranteed (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 safety, from safest to least safe, these are

- **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 always 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 the least safe, and 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 needed. 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

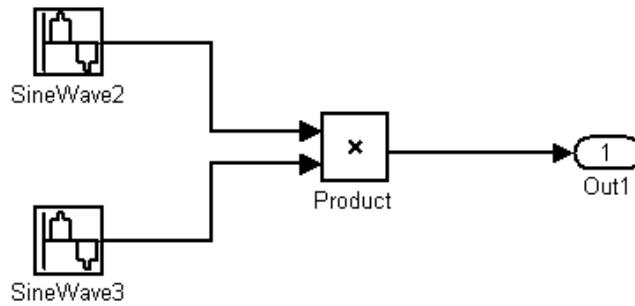
The Simulink engine can detect mismatched rate transitions in a multitasking model and automatically insert Rate Transition blocks to handle them. To instruct the engine to do this, select **Automatically handle rate transition**

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,

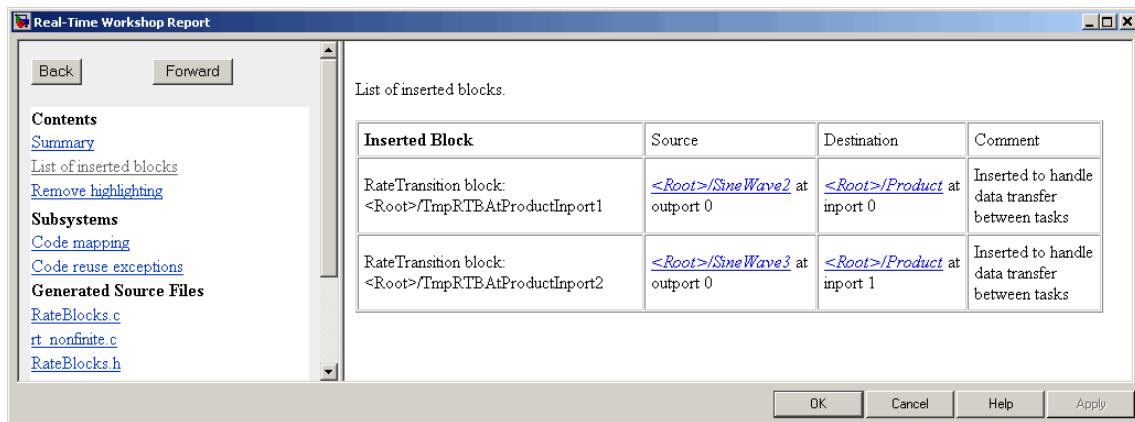
- The Simulink engine handles all transitions between periodic sample times and asynchronous tasks.
- The Simulink engine inserts “hidden” Rate Transition blocks that are not visible on the block diagram.
- The Real-Time Workshop software generates code for the automatically inserted Rate Transition 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 ensures 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 necessary to reconcile the Sine Wave block sample times.

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**.

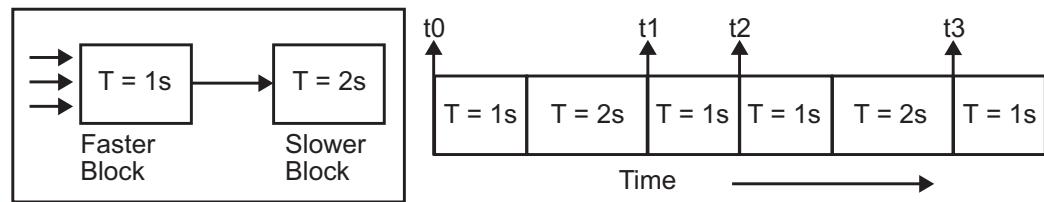
Rate Transition Blocks and Continuous Time

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 necessary 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

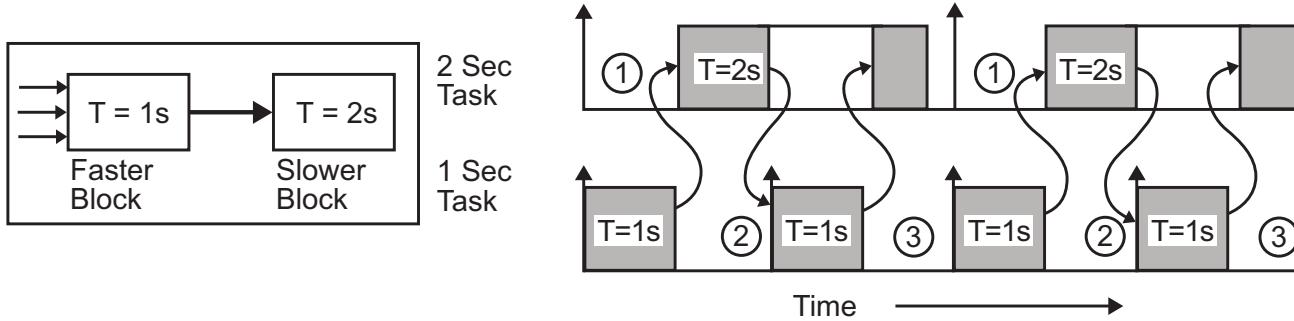
In a model where a faster block drives a slower block having direct feedthrough, the outputs of the faster block are always computed first. In simulation intervals where the slower block does not execute, the simulation progresses more rapidly because there are fewer blocks to execute. The next figure 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 necessary to complete simulation flow. The actual time interval between sample time steps can vary.

Faster to Slower Transitions in Real Time

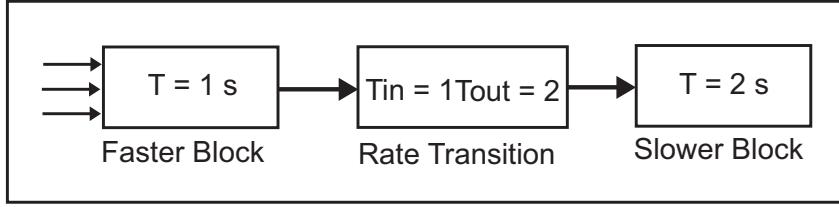
In models where a faster block drives a slower block, you must compensate for the fact that execution of the slower block might span more than one execution period of the faster block. This means that the outputs of the faster block can change before the slower block has finished computing its outputs.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



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

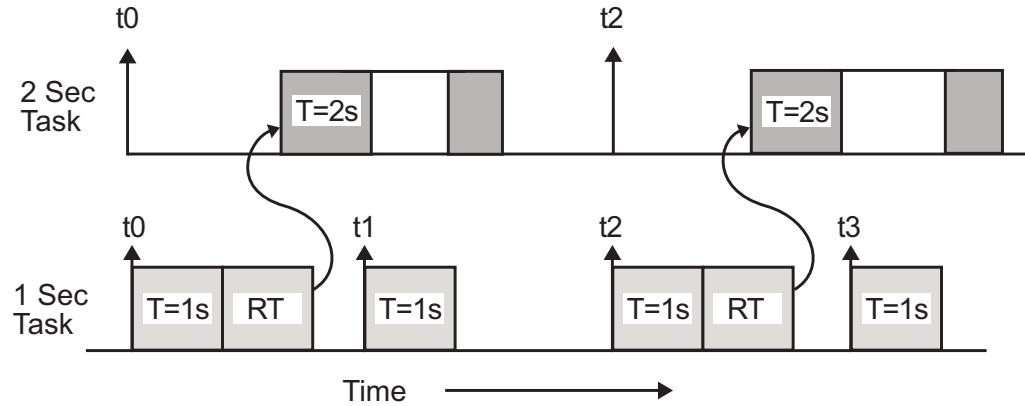
In the above figure, the faster block executes a second time before the slower block has completed execution. This can cause unpredictable results because the input data to the slow task is changing. Data integrity is not guaranteed in this situation.

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. This guarantees that the input to the slower block does not change during its execution, ensuring 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.

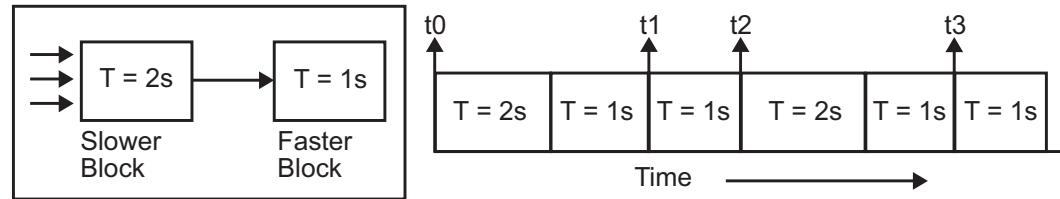


Adding a Rate Transition block ensures that the Rate Transition block executes before the 2 second block (its priority is higher) and that its output value is held constant while the 2 second block executes (it executes at the slower sample rate).

Slower to Faster Transitions in a Simulink Model

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

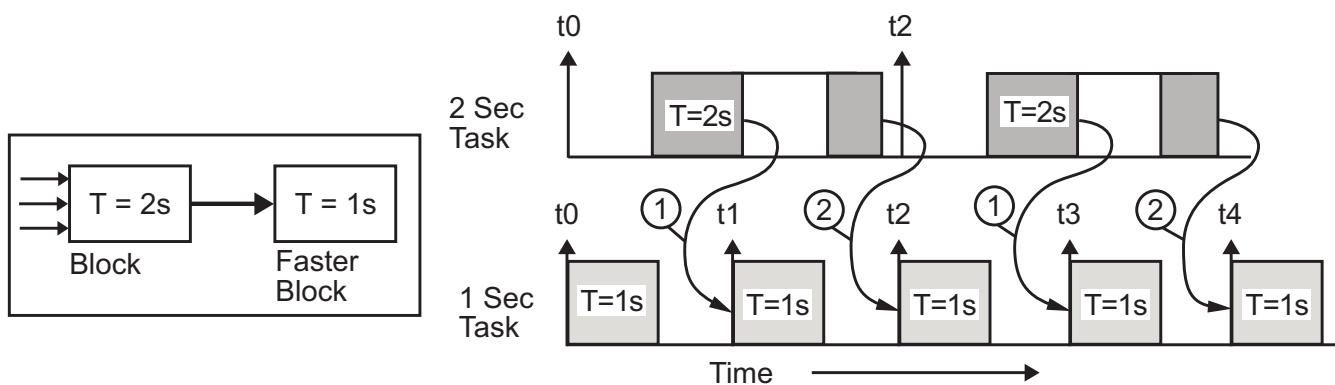
The next figure shows the execution sequence.



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

Slower to Faster Transitions in Real Time

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

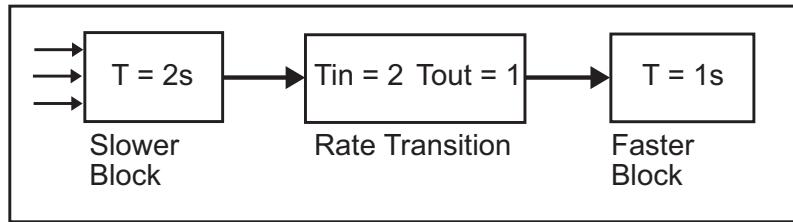


- ① The faster block executes a second time prior to the completion of the slower block.
- ② The faster block executes before the slower block.

This timing diagram illustrates two problems:

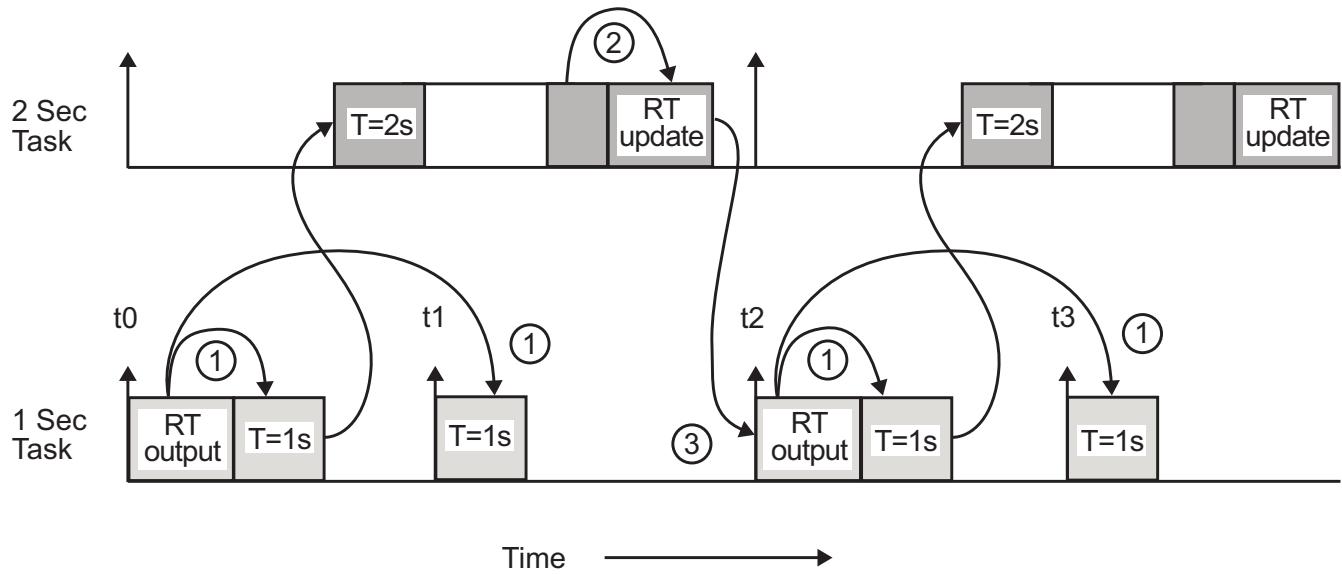
- Execution of the slower block is split over more than one faster block interval. In this case the faster task executes a second time before the slower task has completed execution. This means the inputs to the faster task can have incorrect values some of the time.
- The faster block executes before the slower block (which is backward from the way a Simulink simulation operates). In this case, the 1 second block executes first; but the inputs to the faster task have not been computed. This can cause unpredictable results.

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



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

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



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

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

- 3 The Rate Transition output in the 1 second task uses the state of the Rate Transition that was updated in the 2 second task.

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

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

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

Example: Single-Tasking and Multitasking Execution of a Model

In this section...

- “Introduction” on page 5-27
- “Single-Tasking Execution” on page 5-28
- “Multitasking Execution” on page 5-30

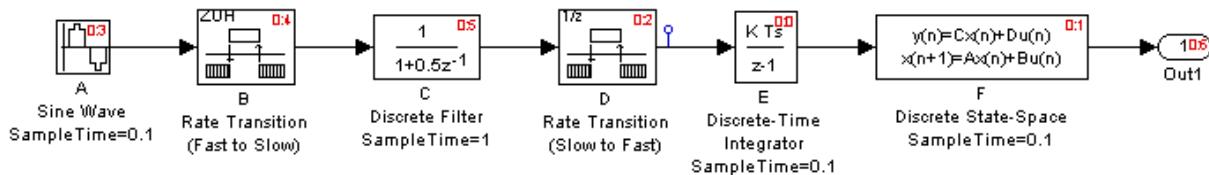
Introduction

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 about block reduction, see “Reducing the Number of Blocks in a Model” on page 26-28.

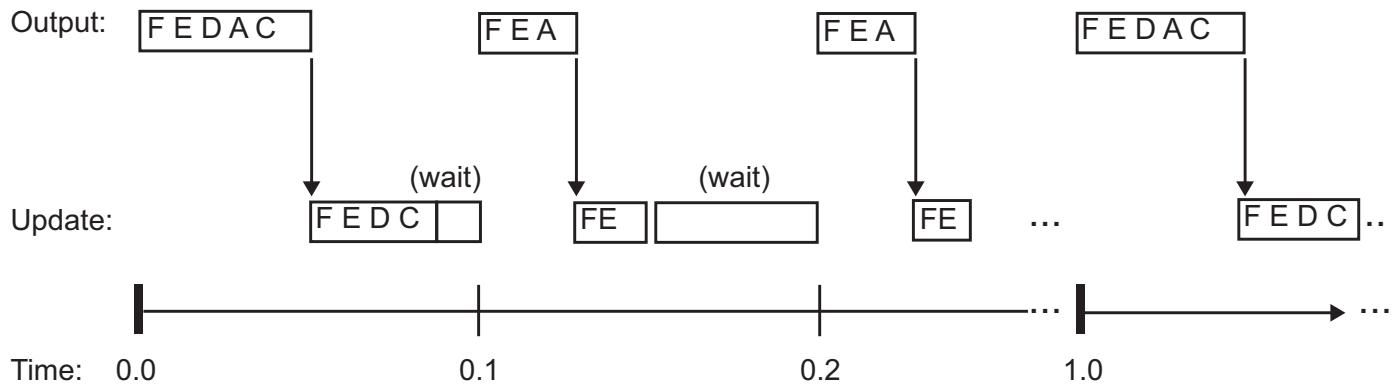
The following table shows, for each block in the model, the execution order, sample time, and whether the block has an output or update computation. Block A does not have discrete states, and accordingly does not have an update computation.

Execution Order and Sample Times (Single-Tasking)

Blocks (in Execution Order)	Sample Time (in Seconds)	Output	Update
F	0.1	Y	Y
E	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.



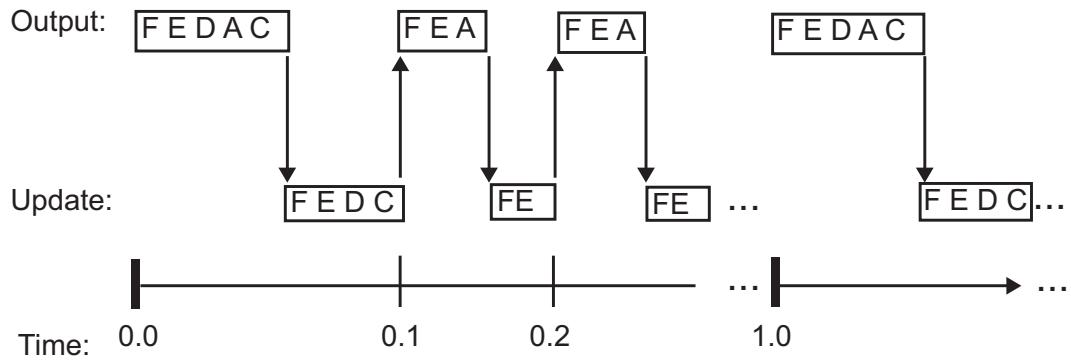
At time 0.0, 1.0, and every second thereafter, both the slow and fast blocks execute their output computations; this is followed by update computations for blocks that have states. Within a given time interval, output and update computations are sequenced in block execution order.

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.

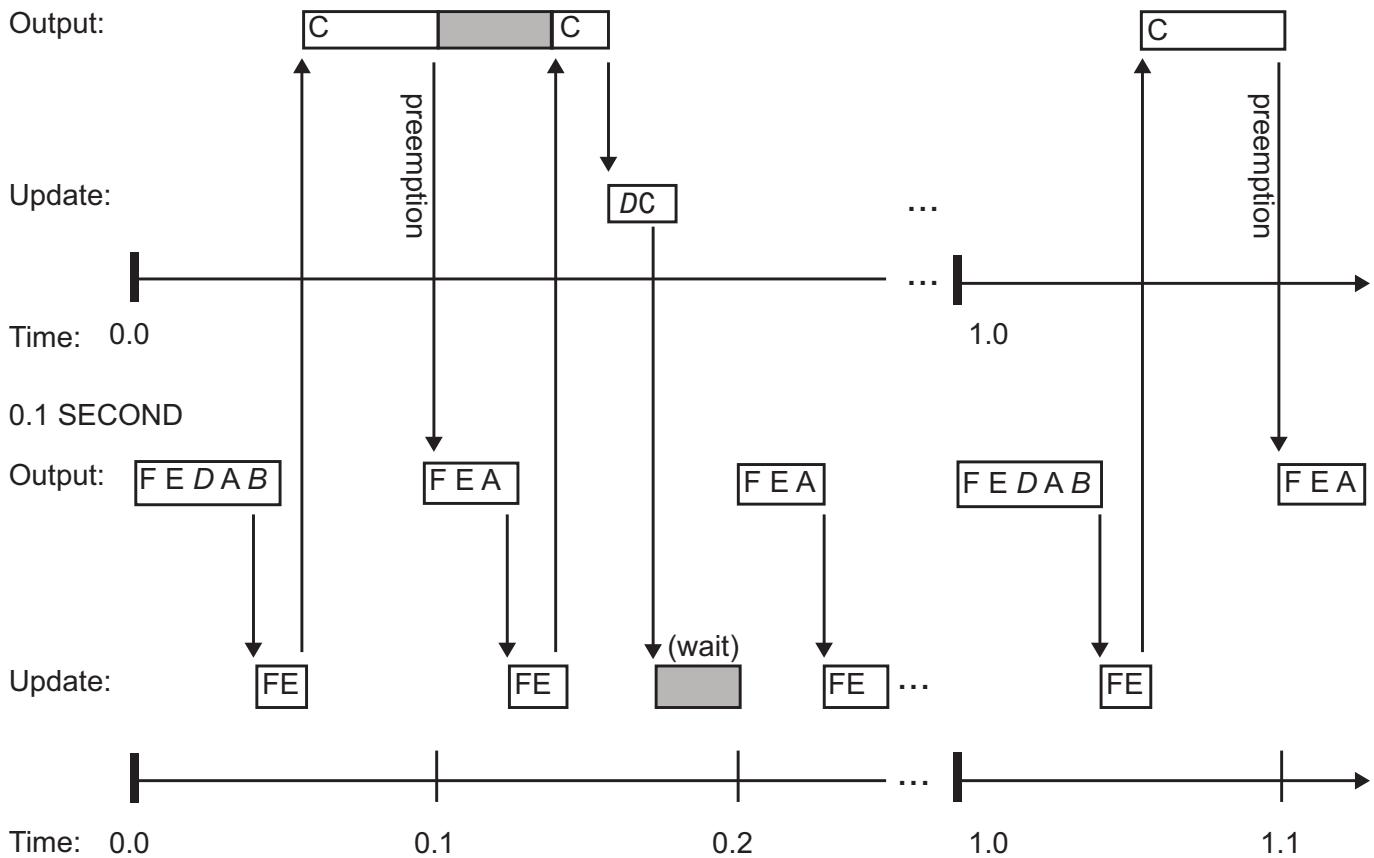
Task Allocation of Blocks in Multitasking Execution

Blocks (in Execution Order)	Task	Output	Update
F	0.1 second task	Y	Y
E	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 “Inheriting 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. For more information on port-based sample times, see “Inheriting Sample Times” in the Simulink documentation.	Y	N
C	1 second task	Y	Y

Real-Time Multitasking Execution

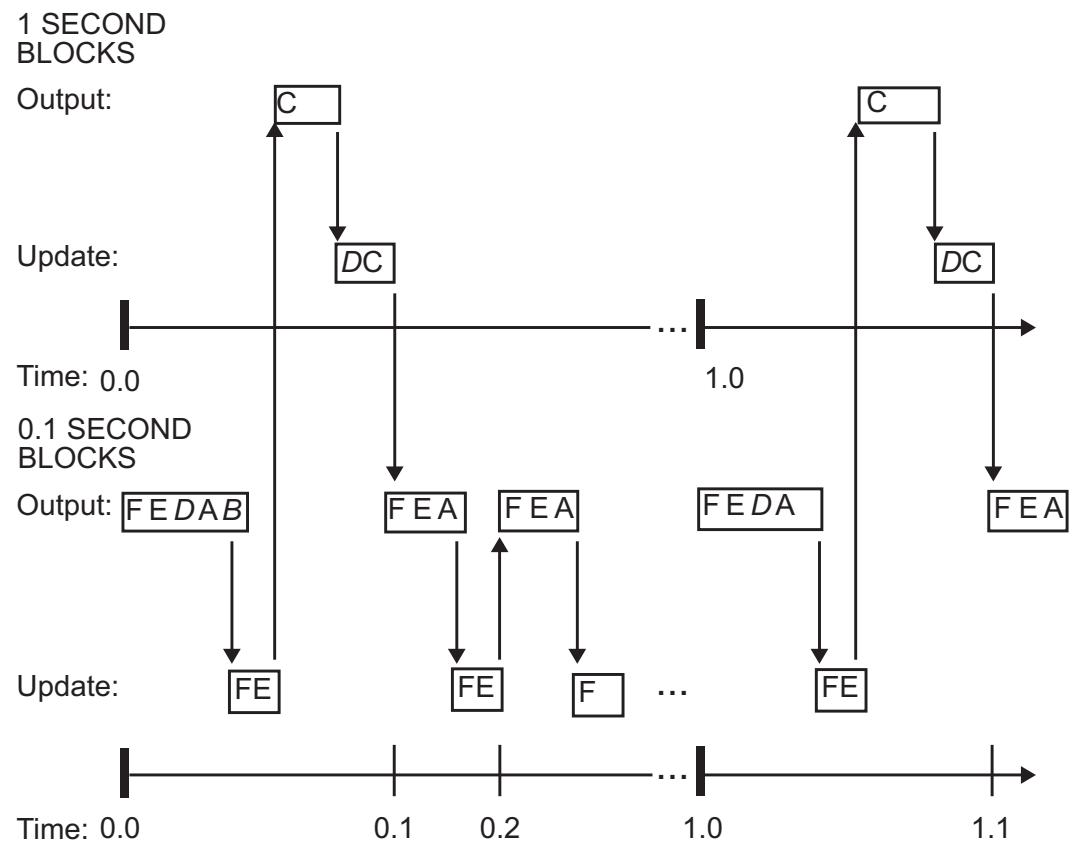
The next figure shows the scheduling of computations in MultiTasking solver mode when the generated code is deployed in a real-time system. The generated program is shown running in real time, as two tasks under control of interrupts from a 10 Hz timer.

1 SECOND



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 all blocks in one thread of execution, simulating multitasking. No preemption occurs.



Handling Asynchronous Events

In this section...

- “Introduction” on page 5-34
- “Handling Interrupts” on page 5-37
- “Rate Transitions and Asynchronous Blocks” on page 5-52
- “Using Timers in Asynchronous Tasks” on page 5-57
- “Creating a Customized Asynchronous Library” on page 5-60
- “Asynchronous Support Limitations” on page 5-69

Introduction

- “About Asynchronous Support” on page 5-34
- “Overview of Block Library for Wind River Systems VxWorks Real-Time Operating System” on page 5-35
- “Accessing the VxWorks Block Library” on page 5-36
- “Generating Code with the VxWorks Library Blocks” on page 5-36
- “Demos and Additional Information” on page 5-36

About Asynchronous Support

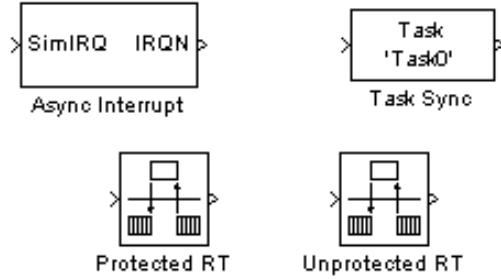
Real-Time Workshop models are normally timed from a *periodic* interrupt source (for example, a hardware timer). Blocks in a periodically clocked single-rate model run at a timer interrupt rate (the base rate of the model). Blocks in a periodically clocked multirate model run at the base rate or at submultiples 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 appropriately, 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 Systems VxWorks Tornado® 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.

Overview of Block Library for Wind River Systems 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 VxWorks Tornado operating system. You can use them, without modification, to support VxWorks applications.

If you want to implement asynchronous support for an RTOS other than 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 “Creating a Customized Asynchronous Library” on page 5-60.

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
- Ensure 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

For detailed descriptions of the blocks in the VxWorks library, see the *Real-Time Workshop Reference*. The use of the Protected and Unprotected Rate Transition blocks in asynchronous contexts is discussed in “Rate Transitions and Asynchronous Blocks” on page 5-52. For general information on rate transitions, see Chapter 5, “Scheduling Considerations”.

Accessing the VxWorks Block Library

The VxWorks library (`vxlib1`) is part of the Real-Time Workshop library. You can access the VxWorks library by opening the Simulink Library Browser, clicking the **Real-Time Workshop** entry, and clicking **VxWorks**. Alternatively, enter the MATLAB command `vxlib1`.

Generating Code with the VxWorks Library Blocks

To generate a VxWorks compatible application from a model containing VxWorks library blocks, select one of the following targets from the System Target File Browser associated with the model:

- `ert.tlc` **Real-Time Embedded Coder**. This target is provided with the Real-Time Workshop 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` **Tornado (VxWorks) Real-Time Target**. This target is included with the Real-Time Workshop product (see Chapter 34, “Interfacing With a Real-Time Operating System”).

Demos and Additional Information

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

- The `rtwdemo_async` model. To open this demo, type `rtwdemo_async` at the MATLAB command prompt.
- Chapter 5, “Scheduling Considerations”, discusses general multitasking and rate transition issues for periodic models.
- Chapter 34, “Interfacing With a Real-Time Operating System”, discusses the Tornado (VxWorks RTOS) target example.
- The Real-Time Workshop 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

- “Generating Interrupt Service Routines” on page 5-37
- “Spawning a Wind River Systems VxWorks Task” on page 5-45

Generating Interrupt Service Routines

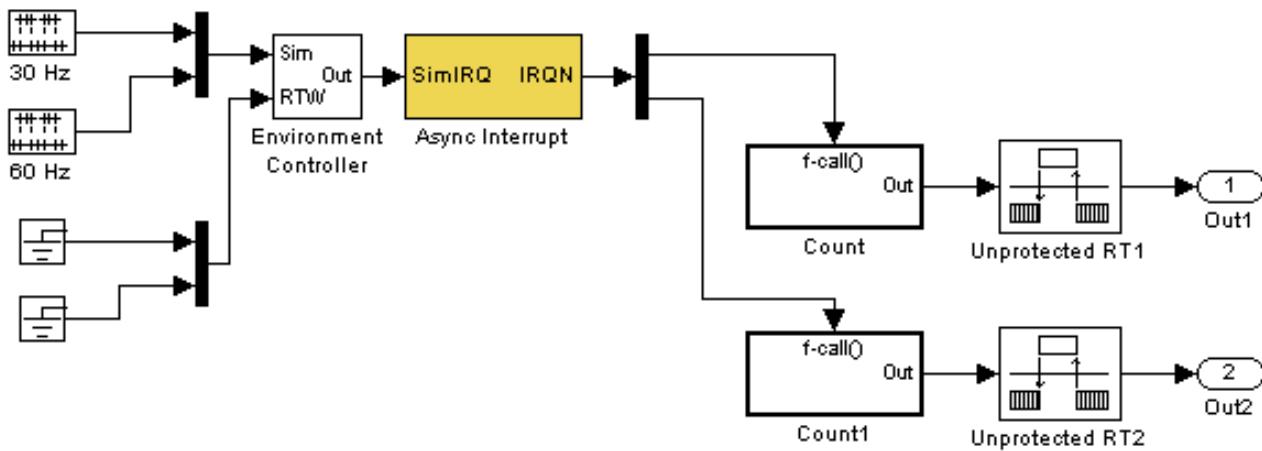
To generate an interrupt service routine (ISR) associated with a specific Wind River Systems VxWorks VME interrupt level, use the Async Interrupt block. The Async Interrupt block enables the specified interrupt level and installs an ISR that calls a connected function call subsystem.

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:

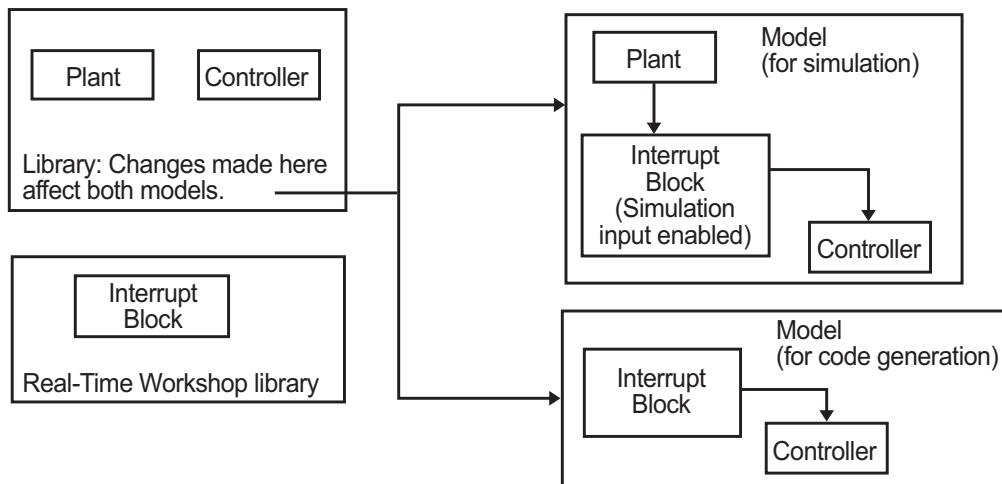
- The Async Interrupt block supports VME interrupts 1 through 7.
- The Async Interrupt block requires a VxWorks Board Support Package (BSP) that supports the following VxWorks system calls:
 - sysIntEnable
 - sysIntDisable
 - intConnect
 - intLock
 - intUnlock
 - tickGet

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 “Spawning a Wind River Systems VxWorks Task” on page 5-45 for more information.

Using the Async Interrupt Block in Simulation and Code Generation.

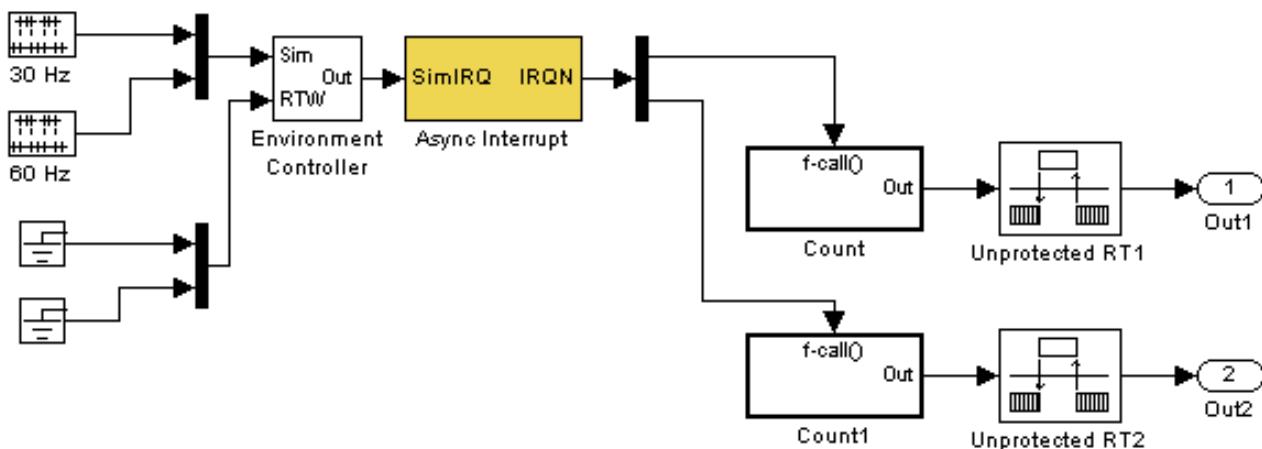
This section describes a *dual-model* approach to the development and implementation of real-time systems that include ISRs. In this approach, you develop one model that includes a plant and a controller for simulation, and another model that only includes the controller for code generation. Using a 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.

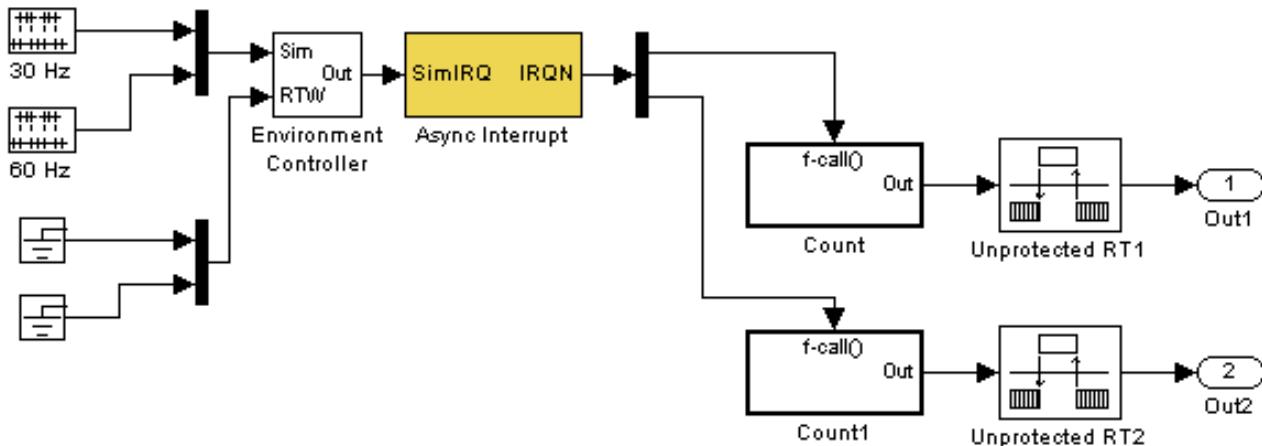
You can also use the Async Interrupt block in a simulation without enabling the simulation input. In such a case, the Async Interrupt block inherits the base rate of the model and calls the connected subsystems in order of their

VxWorks priorities. (In effect, 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 to ensure that no code is generated for that signal path. The Real-Time Workshop 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, ensure that the sample times of the blocks driving the simulation input are used in the model where generated code is intended.

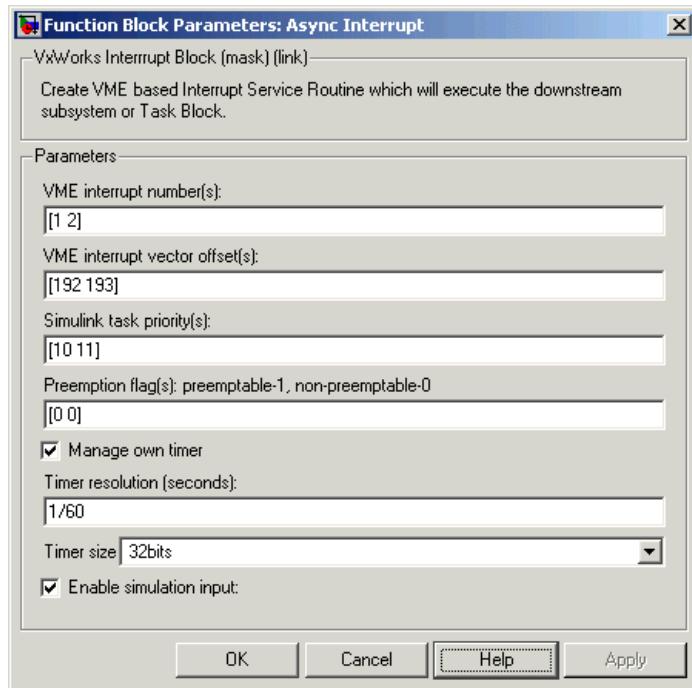


Standalone functions are installed as ISRs and the interrupt vector table is as follows:

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 substrate 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 “Using Timers in Asynchronous Tasks” on page 5-57 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);
}
```

Spawning a Wind River Systems 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.

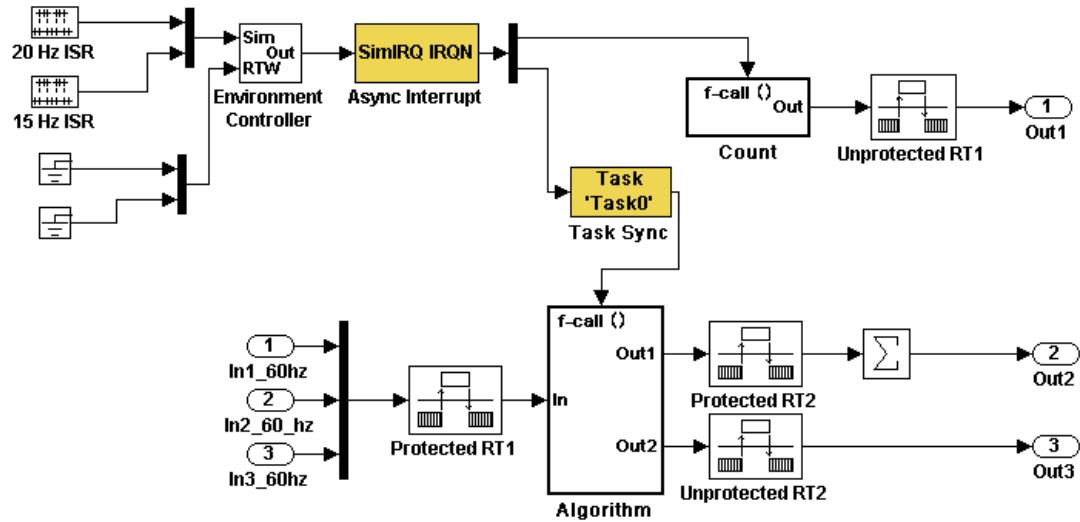
The Task Sync block performs the following functions:

- 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.

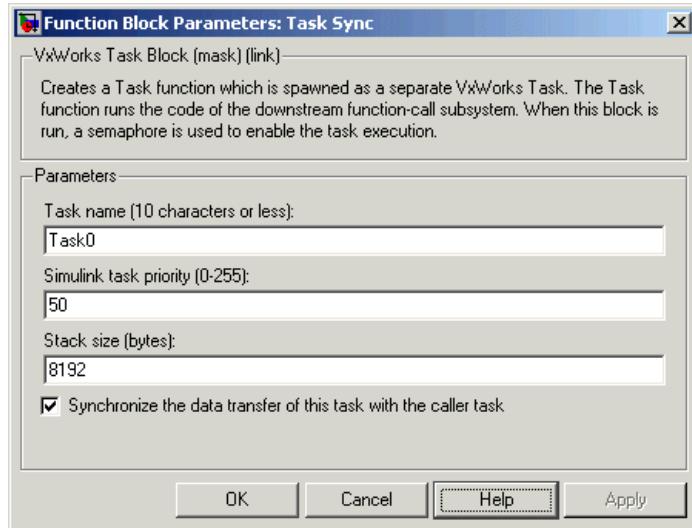
- The Task Sync block generates synchronization code (for example, `semGive()`). This code allows the spawned task to run; the task in turn calls the connected function call subsystem code. The synchronization code can run at interrupt level. This is accomplished by connecting the Task Sync block to 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` demo. The block diagram for the model appears in the next figure. Before reading the following discussion, open the demo model and double-click the **Generate Code** button. You can then examine the generated code in the HTML code generation report produced by the demo.



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: <$5>/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)) == 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 demo, 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.clockTick3`) is updated with the value of the timer that is maintained by the Async Interrupt block (`rtM->Timing.clockTick4`). 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) */
/* Spawns 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];
}
}
}

```

5 Scheduling Considerations

```
}

{

    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 *
    1.666666666666666E-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

- “Introduction” on page 5-52
- “Handling Rate Transitions for Asynchronous Tasks” on page 5-54
- “Handling Multiple Asynchronous Interrupts” on page 5-55

Introduction

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 Chapter 5, “Scheduling Considerations”.

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 the inserted blocks to ensure data integrity but not determinism during data transfers.

For asynchronous rate transitions, the Rate Transition block guarantees data integrity, but cannot guarantee 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 ensure data integrity and priorities are assigned to the tasks associated with the blocks, the Real-Time Workshop 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 Real-Time Workshop 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 Systems 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).

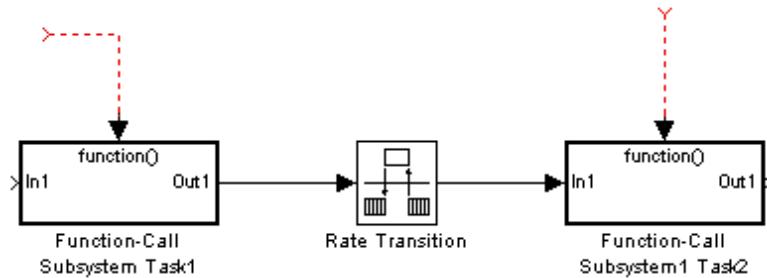
The VxWorks library provides two types of rate transition blocks as a convenience. These are simply preconfigured instances of the built-in Simulink Rate Transition block:

- 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.

Handling Rate Transitions for Asynchronous Tasks

For rate transitions that involve asynchronous tasks, you can ensure data integrity. However, you cannot ensure 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:

- Ensure data integrity, no determinism
- Unprotected

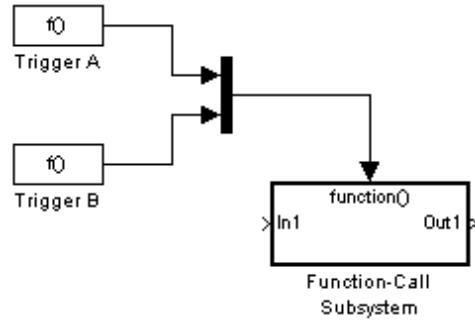
Alternatively, you can use target-specific rate transition blocks. The following blocks are available for the VxWorks RTOS:

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

- Unprotected Rate Transition block

Handling 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)		

**Supported Sample Time and Priority for Function Call Subsystem with Multiple Triggers
(Continued)**

	Async Priority = 1	Async Priority = 2	Async Priority Unspecified	Periodic Priority = 1	Periodic Priority = 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 Real-Time Workshop 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 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 Real-Time Workshop code generator generates timer code in the function call subsystem, updating the clock tick variable from the base rate clock tick. Data integrity is ensured 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 “Using Timers in Asynchronous Tasks” on page 5-57 and “Controlling Memory Allocation for Time Counters” on page 26-51 for more information.

Using 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 any 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 any 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 any 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 Systems 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 the appropriate timer register on the target hardware.

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

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

For an example of timer code generation, see “[Async Interrupt Block Implementation](#)” on page 5-61.

Creating a Customized Asynchronous Library

- “Introduction” on page 5-60
- “Async Interrupt Block Implementation” on page 5-61
- “Task Sync Block Implementation” on page 5-65
- “`asynclib.tlc` Support Library” on page 5-67

Introduction

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 Systems 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:

- The “Overview of S-Functions” in the Simulink Writing S-Functions documentation describes MEX S-functions and the S-function API in general.
- The Target Language Compiler documentation and Chapter 32, “Integrating External Code Using S-Functions” 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 5-67.

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 any one of the above macros is called before `ssSetOutputPortWidth`, the following error message will appear:

```
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.

Generating #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, there is no requirement to use this naming convention.

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

Outputs Function

`Outputs` iterates over all 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 appropriate counter. In your implementation, you should generate either an equivalent call to the target RTOS, or generate code to read the appropriate 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 appropriate calls to your target RTOS.

Terminate Function

The `Terminate` function generates the call `sysIntDisable` to disable each ISR. You should replace this with appropriate 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 5-67.

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.

Generating #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 appropriate 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 “Spawning a Wind River Systems VxWorks Task” on page 5-45. `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 appropriate calls to your target RTOS.

Note also that if the target RTOS has dynamically allocated any memory associated with the task , the `Terminate` function should deallocate the memory.

asynlib.tlc Support Library

`asynlib.tlc` is a library of TLC functions that support the implementation of asynchronous blocks. Some functions are specifically designed for use in asynchronous blocks. For example, `LibSetAsyncCounter` generates a call to update a timer for an asynchronous block. Other functions are utilities that

return information required by asynchronous blocks (for example, information about connected function call subsystems).

The following table summarizes the public calls in the library. For details, see the library source code and the `vxinterrupt1.tlc` and `vxtask1.tlc` files, which call the library functions.

Summary of `asynclib.tlc` Library Functions

Function	Description
<code>LibGetBlockAttribute</code>	Returns a field value from a block record.
<code>LibGetFcnCallBlock</code>	Given an S-Function block and call index, returns the block record for the downstream function call subsystem block.
<code>LibBlockExecuteFcnCall</code>	For use by inlined S-functions with function call outputs. Generates code to execute a function call subsystem. <code>LibBlockExecuteFcnCall</code> calls the lower-level function <code>LibExecuteFcnCall</code> , but has a simplified argument list. See the Target Language Compiler documentation for more information on <code>LibExecuteFcnCall</code> .
<code>LibGetCallerClockTickCounter</code>	Provides access to an upstream asynchronous task's time counter.
<code>LibGetCallerClockTickCounterHighWord</code>	Provides access to the high word of an upstream asynchronous task's time counter.
<code>LibManageAsyncCounter</code>	Determines whether an asynchronous task needs a counter and manages its own timer.

Summary of `asyolib.tlc` Library Functions (Continued)

Function	Description
<code>LibNeedAsyncCounter</code>	If the calling block requires an asynchronous counter, returns <code>TLC_TRUE</code> , otherwise returns <code>TLC_FALSE</code> .
<code>LibSetAsyncClockTicks</code>	Returns code that sets <code>clockTick</code> counters that are to be maintained by the asynchronous task.
<code>LibSetAsyncCounter</code>	Generates code to set the tick value of the block's asynchronous counter.
<code>LibSetAsyncCounterHighWord</code>	Generates code to set the tick value of the high word of the block's asynchronous counter

Asynchronous Support Limitations

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.

Using Timers

In this section...

“Absolute and Elapsed Time Computation” on page 5-70

“APIs for Accessing Timers” on page 5-72

“Elapsed Timer Code Generation Example” on page 5-77

“Limitations on the Use of Absolute Time” on page 5-80

Absolute and Elapsed Time Computation

- “Introduction” on page 5-70
- “Timers for Periodic and Asynchronous Tasks” on page 5-71
- “Allocation of Timers” on page 5-71
- “Integer Timers in Generated Code” on page 5-72
- “Elapsed Time Counters in Triggered Subsystems” on page 5-72

Introduction

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). All 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, on an as-needed basis. If no blocks executing at a given rate require a timer, no timer is 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 Real-Time Workshop product provides S-function and TLC APIs that let your S-functions access timers, in both simulation and code generation.
- For ERT and ERT-derived targets, the word size of the timers is determined by a user-specified maximum counter value. Correct specification of this value ensures that timers will not overflow. See the description of the parameter “Controlling Memory Allocation for Time Counters” on page 26-51. See also the Real-Time Workshop Embedded Coder documentation for information on restrictions on its use.

See “Limitations on the Use of Absolute Time” on page 5-80 and “Blocks that Depend on Absolute Time” on page 5-81 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 Real-Time Workshop 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:

- “Using Timers in Asynchronous Tasks” on page 5-57
- “Creating a Customized Asynchronous Library” on page 5-60

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 5-72). 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 “Using Timers in Asynchronous Tasks” on page 5-57 and “Controlling Memory Allocation for Time Counters” on page 26-51 for more information.

Elapsed Time Counters in Triggered Subsystems

Some blocks, such as the Discrete-Time Integrator block, perform computations requiring the elapsed time (ΔT) since the previous block execution. Blocks requiring elapsed time data must register the requirement (see “APIs for Accessing Timers” on page 5-72). 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 any 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 always 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

- “Introduction” on page 5-73

- “C API for S-Functions” on page 5-73
- “TLC API for Code Generation” on page 5-76

Introduction

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 correctly, 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 Writing S-Functions 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 5-72.
- `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;
.
.
```

```
ssGetElapsedTime(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.

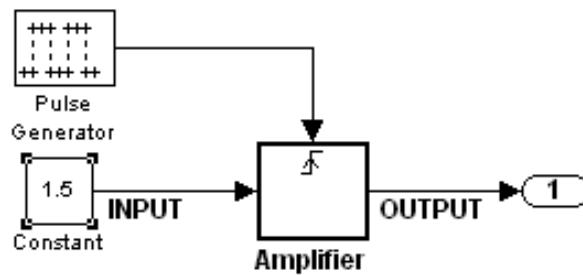
`LibGetTaskTimeFromTID` is documented with other sample time functions in the TLC Function Library Reference pages of the Target Language Compiler documentation.

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 substrate, the wrong timer is read, causing serious errors.

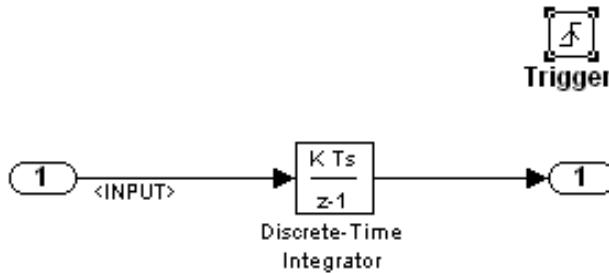
- `LibGetElapsedTime(system)`: Generates code to read the elapsed time counter for `system`. (`system` is the parent system of the calling block.) See “Elapsed Timer Code Generation Example” on page 5-77 for an example of code generated by this function.
- `LibGetElapsedTimeCounter(system)`: Generates 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 `LibGetElapsedTimeCounterDtypeId` and `LibGetElapsedTimeResolution`. (See the discussion of `ssGetElapsedTimeCounter` above.)
- `LibGetElapsedTimeCounterDtypeId(system)`: Generates code that returns the data type of the elapsed time counter for `system`. (`system` is the parent system of the calling block.)
- `LibGetElapsedTimeResolution(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 clockTickCount;                /* '<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' */
```

```
elapsedTime_exp_DWork.DiscreteTimeIntegrator_DSTATE += 0.3 *  
(real_T)rt_elapsedTime * 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 `LibGetElapsedTime` to access the elapsed time value.

Limitations on the Use of Absolute Time

- “About Absolute Time Limitations” on page 5-80
- “Logging Absolute Time” on page 5-80
- “Absolute Time in Stateflow Charts” on page 5-81
- “Blocks that Depend on Absolute Time” on page 5-81

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 5-70 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 no longer correct.

If your target uses `rtModel`, you can avoid timer overflow by setting an appropriate **Application life span** option. See “Integer Timers in Generated Code” on page 5-72 for more information.

Logging Absolute Time

If you log time values by enabling **Configuration Parameters > 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.

Configuring Scheduling

In this section...

[“Configuring Start and Stop Times” on page 5-83](#)

[“Configuring the Solver Type” on page 5-84](#)

[“Configuring the Tasking Mode” on page 5-84](#)

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

Configuring 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 Systems Tornado targets, you can override the stop time when running a generated program from the Microsoft Windows command prompt or UNIX³ 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. See Getting Started in the Real-Time Workshop documentation for an example of the use of this option.

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 5-80 for a list of blocks that can potentially overflow timers.

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

If you know how long an application that depends on absolute time needs to run, you can ensure that timers do not overflow and that they use optimal word sizes by specifying the **Application lifespan (days)** parameter on the **Optimization** pane. See “Controlling Memory Allocation for Time Counters” on page 26-51 for details.

Configuring 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.

Configuring the Tasking Mode

The Real-Time Workshop product supports both single-tasking and multitasking modes for periodic sample times. See Chapter 5, “Scheduling Considerations” for details.

Simulation Considerations That Affect Code Generation

- “Overview” on page 6-2
- “Sample Time Propagation” on page 6-4
- “Latches for Subsystem Blocks” on page 6-6
- “Block Execution Order” on page 6-7
- “Algebraic Loops” on page 6-9

Overview

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

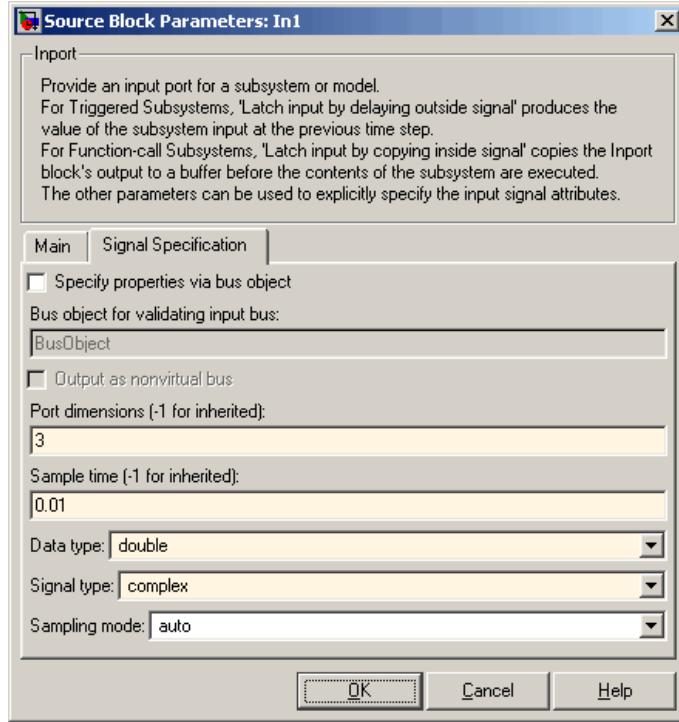
The first stage of code generation is compilation of the block diagram. This stage is analogous to that of a C or C++ program. The compiler carries out type checking and preprocessing. Similarly, the Simulink engine verifies that input/output data types of block ports are consistent, line widths between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

You can verify what data types any given Simulink block supports by typing

```
showblockdatatypetable
```

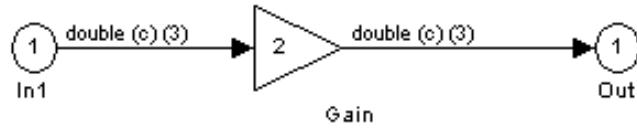
at the MATLAB prompt, or (from the Help browser) clicking the command above.

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



In this example, the Import 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 Import block through a simple block diagram.



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

Sample Time Propagation

Inherited sample times in source blocks (for example, a root import) 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.

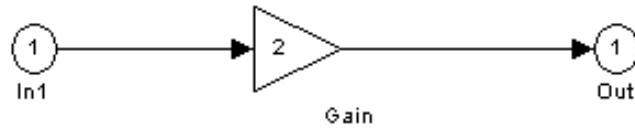
In such cases, the Simulink engine propagates the known or assigned sample times to those blocks that have inherited sample times but that have not yet been assigned a sample time. Thus, the engine continues to fill in the blanks (the unknown sample times) until sample times have been assigned to as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

Blocks that still do not have a sample time are assigned a default sample time according to the following rules:

- 1** If the current system has at least one rate in it, the block is assigned the fastest rate.
- 2** If no rate exists and the model is configured for a variable-step solver, the block is assigned a continuous sample time (but fixed in minor time steps). The Real-Time Workshop product (with the exception of the rapid simulation and S-function targets) does not currently support variable-step solvers.
- 3** If no rate exists and the model is configured for a fixed-step solver, the block is assigned a discrete sample time of $(T_f - T_i)/50$, where T_i is the simulation start time and T_f is the simulation stop time. If T_f is infinity, the default sample time is set to 0.2.

To ensure a completely deterministic model (one where no sample times are set using the above rules), you should explicitly specify the sample times of all your source blocks. Source blocks include root import blocks and any 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 default sample time is derived from destination blocks or the default rule.



All blocks have an inherited sample time ($T_s = -1$). They are all 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 any 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 Import 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 Import 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 Real-Time Workshop code generator

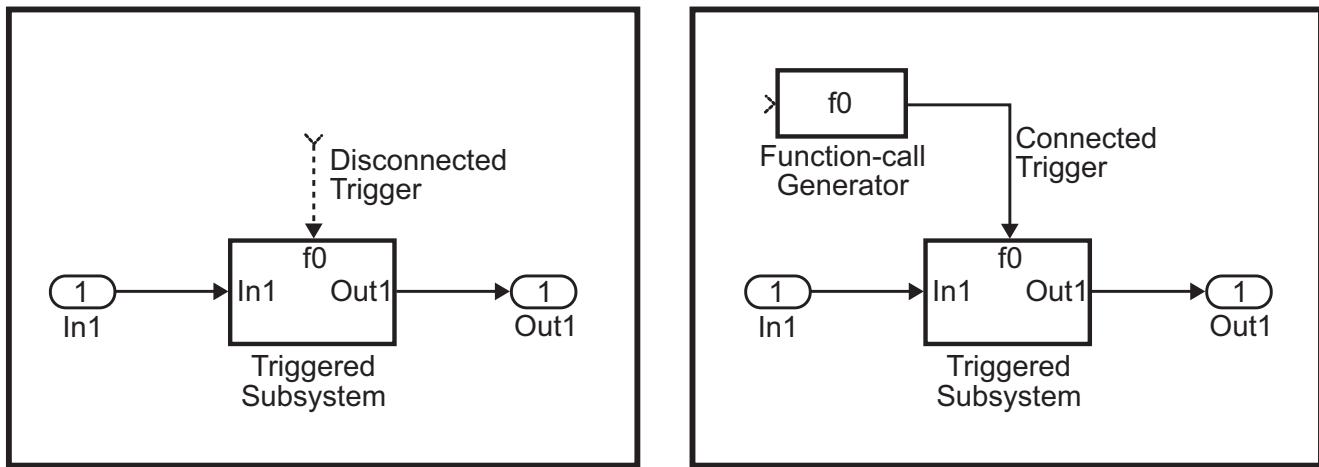
- Preserves latches in generated code regardless of any optimizations that might be set
- Places the code for latches at the start of a subsystem's output/update function

For more information on these options, see the description of the Import 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 all the connection information of the model, as well as the necessary signal attributes. Thus, the timing engine 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 all blocks inheriting a constant sample time. Calling the trigger function, `f()`, directly from user code does not work correctly and should never be done. Instead, you should use a function-call generator to properly 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 any other 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 Real-Time Workshop 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 Real-Time Workshop products ensure 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 Real-Time Workshop libraries, see “Handling Asynchronous Events” on page 5-34. For more information on multirate code generation, see Chapter 5, “Scheduling Considerations”.

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

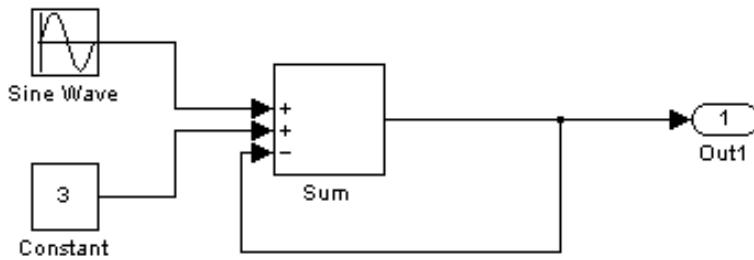
- $x = y + 2$
- $y = -x$

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

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

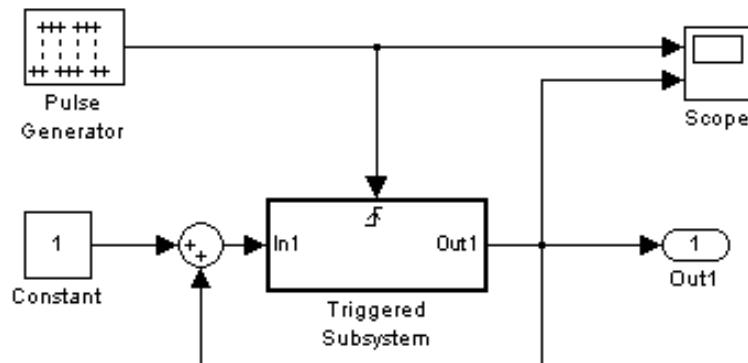
An algebraic loop exists whenever the output of a block having direct feedthrough (such as Gain, Sum, Product, and Transfer Fcn) is fed back as an input to the same block. The Simulink engine is often able to solve models that contain algebraic loops, such as the next diagram.



The Real-Time Workshop 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 all or some 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 necessary and feasible. If you want the Real-Time Workshop 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 always 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 import in the above triggered subsystem, the Simulink engine cannot solve the problem, and the Real-Time Workshop 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 Real-Time Workshop software to generate code for models containing Model blocks that are involved in algebraic loops.

Protecting Intellectual Property in Shared Model Components

Real-Time Workshop technology supports two approaches for protecting the intellectual property of your designs and algorithms in code that is generated to be shared.

- Use the protected model, a referenced model from which all block and line information has been eliminated using the Model Protection facility. For more information, see “Protecting Referenced Models” in the Simulink documentation
- Use the Real-Time Workshop Embedded Coder shared library system target file to generate a shared library for a model or subsystem for use in a system simulation external to Simulink. For more information see “Creating and Using Host-Based Shared Libraries” in the Real-Time Workshop Embedded Coder documentation.

You can then deploy the protected model or shared library without sharing the model itself or its generated source code.

Stateflow Considerations for Event-Driven Applications

If your application model uses Stateflow software to simulate events and you have a Stateflow Coder product license installed, consider the following information:

- You can configure a model so that the code generator includes:
 - Stateflow object comments and descriptions in generated code. See “Simulink block / Stateflow object comments” and “Stateflow object descriptions”.
 - A summary of Stateflow objects and the corresponding code location in code generation reports for traceability. See “Traceable Stateflow objects”.
 - If you run a Simulink model in external mode, you can animate states and view Stateflow test points in floating scopes and signal viewers. See Chapter 37, “Communicating With Code Executing on a Target System Using Simulink External Mode”.
 - You can generate reusable code from a Stateflow chart, or from a subsystem containing a chart, except in the following cases:
 - The Stateflow chart contains exported graphical functions.
 - The Stateflow model contains events that have a machine parent.
- For more information on generating reusable code, see “Creating Reusable Components” on page 3-73.

Defining Data Representation and Storage for Code Generation

- Chapter 9, “Deciding on Data Representation and Storage”
- Chapter 10, “Parameter Considerations”
- Chapter 11, “Signal Considerations”
- Chapter 12, “Simulink Data Object Considerations”
- Chapter 13, “Enumerated Data Type Considerations”
- Chapter 14, “Block State Storage and Interfacing Considerations”
- Chapter 15, “Data Store Memory Considerations”



Deciding on Data Representation and Storage

After you validate a model and produce a specification, you design the software by using a system simulator or rapid prototyping hardware.

The Real-Time Workshop and Real-Time Workshop Embedded Coder products support the Simulink built-in data types. In addition, you can use the Simulink Fixed Point product to specify fixed-point math for Simulink, Stateflow, and Embedded MATLAB models. Fixed-point processing is especially important in mass production environments that cannot afford microprocessors with floating-point units. The process of converting from floating-point to fixed-point math is time consuming, whether or not you are using models. The Simulink Fixed Point product includes conversion and scaling tools that help automate that process.

The Simulink Fixed Point product offers bit-wise accurate simulation results when compared with the behavior on the actual embedded microprocessor. It does this by using integer word sizes and other hardware characteristics that you specify when you configure the model. You do not need to generate code to perform fixed-point simulation or analysis. However, after assessing the fixed-point model and confirming that the results match those of the specification, or other reference, you can generate code for implementation and further analysis. The Real-Time Workshop Embedded Coder product has specific features that can help you generate highly efficient fixed-point code. For example, you can explicitly suppress the generation of floating-point code.

Parameter Considerations

- “Introduction” on page 10-2
- “Nontunable Parameter Storage” on page 10-3
- “Tunable Parameter Storage” on page 10-6
- “Tunable Parameter Storage Classes” on page 10-8
- “Using the Model Parameter Configuration Dialog Box” on page 10-11
- “Tunable Expressions” on page 10-16
- “Linear Block Parameter Tunability” on page 10-20
- “Parameter Configuration Quick Reference Diagram” on page 10-22
- “Generated Code for Parameter Data Types” on page 10-23
- “Tunable Workspace Parameter Data Type Considerations” on page 10-29
- “Parameter Tuning by Using MATLAB Commands” on page 10-31

Introduction

This section discusses how the Real-Time Workshop 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 “Working with 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 Chapter 35, “Interacting with Target Application Data Using the C API”.

Simulink external mode offers a way to monitor signals and modify parameter values while generated model code executes. However, external mode might not be appropriate for your application in some cases. The S-function target does not support external mode, for example. 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 Chapter 37, “Communicating With Code Executing on a Target System Using Simulink External Mode”.

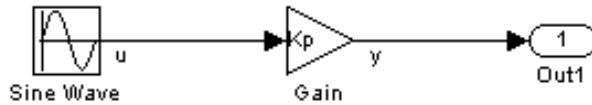
Nontunable Parameter Storage

By default, block parameters are not tunable in the generated code. When **Inline Parameters** is off (the default), the Real-Time Workshop product has control of parameter storage declarations and the symbolic naming of parameters in the generated code.

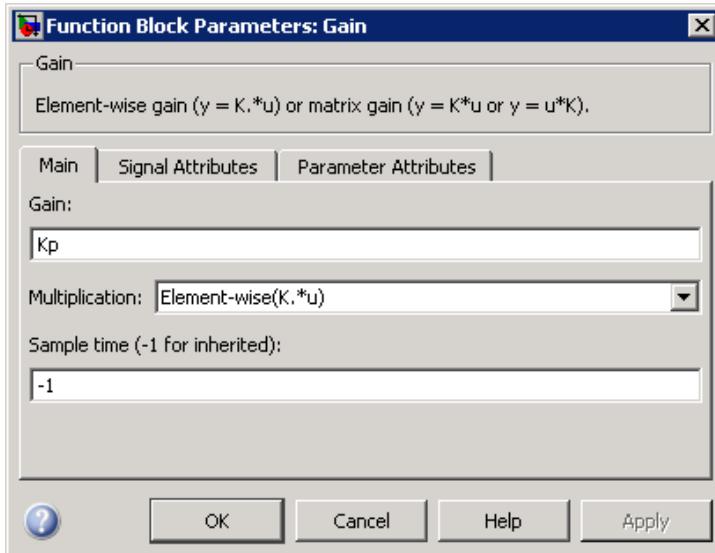
Nontunable parameters are stored as fields within *model_P* (formerly *rtP*), a model-specific global parameter data structure. The Real-Time Workshop product initializes each field of *model_P* to the value of the corresponding block parameter at code generation time.

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

Inline Parameters	Generated Variable Declaration and Code
	<pre> 1.0 , /* SineWave_Freq : '<Root>/Sine Wave' */ 0.0 , /* SineWave_Phase : '<Root>/Sine Wave' */ 5.0 /* Gain_Gain : '<Root>/Gain' */ }; . . . non_tunable_sin_Y.Out1 = rtb_u * non_tunable_sin_P.Gain_Gain;</pre>
On	<pre>non_tunable_sin_Y.Out1 = rtb_u * 5.0;</pre>

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 Real-Time Workshop product declares the parameter in generated code.

The term “storage class,” as used in the Real-Time Workshop 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, without error checking.
- (Implicitly) the symbolic name of the variable or field in which the parameter is stored. The Real-Time Workshop product derives variable and field names from the names of tunable parameters.

The Real-Time Workshop 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 properly 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 10-16.)

Overriding 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 Chapter 12, “Simulink Data Object Considerations” for specific details.

The mechanics of declaring tunable parameters are discussed in “Using the Model Parameter Configuration Dialog Box” on page 10-11.

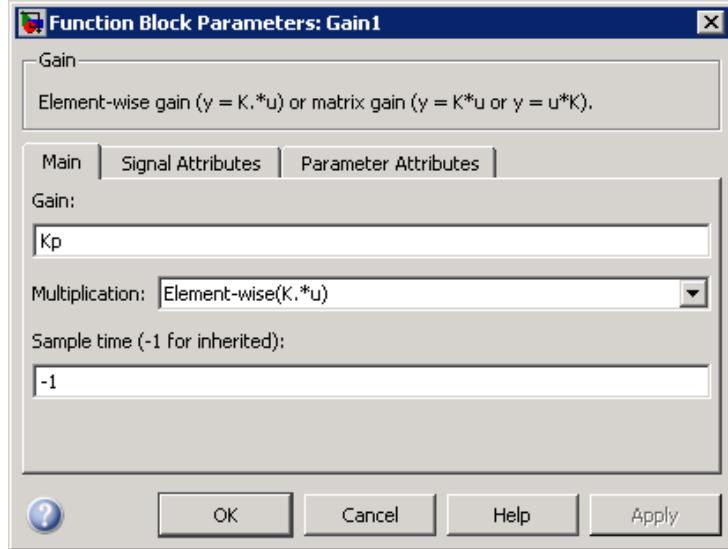
Tunable Parameter Storage Classes

The Real-Time Workshop 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 Real-Time Workshop 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 proper variable definition and initializer.
- **ImportedExternPointer**: `model_private.h` declares the variable as an `extern` pointer. Your code must supply the proper 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.

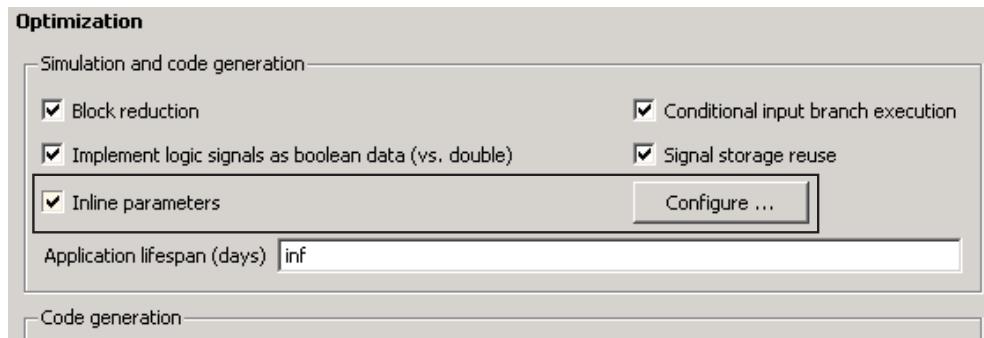
Note The Real-Time Workshop product uses column-major ordering for two-dimensional signal and parameter data. When interfacing your hand-written code to such signals or parameters by using `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` declarations, make sure that your code observes this 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)	<pre> 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; </pre>
ExportedGlobal	<pre> real_T Kp = 3.14; . . tunable_sin_Y.Out1 = rtb_u * Kp; </pre>
ImportedExtern	<pre> extern real_T Kp; . . tunable_sin_Y.Out1 = rtb_u * Kp; </pre>
ImportedExtern Pointer	<pre> extern real_T *Kp; . . tunable_sin_Y.Out1 = rtb_u * (*Kp); </pre>

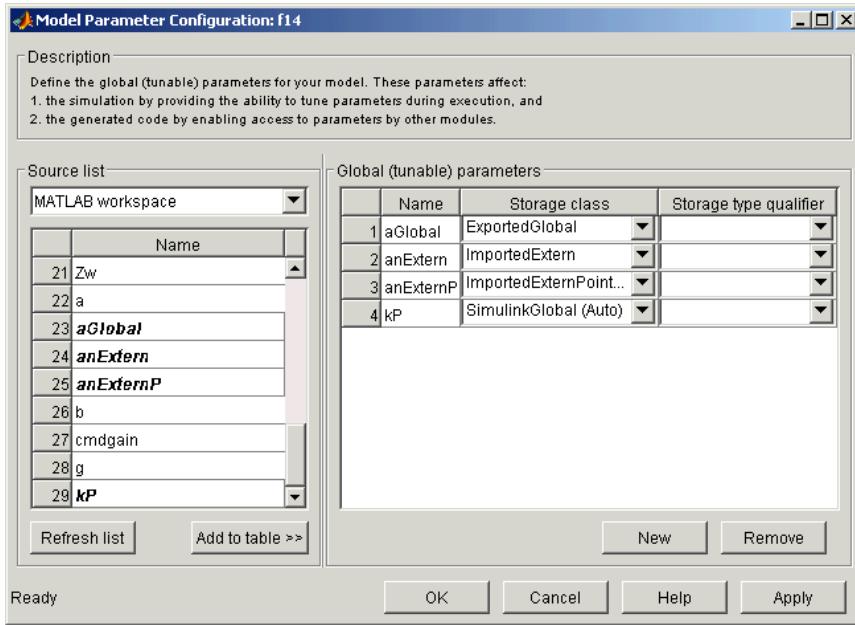
Using the Model Parameter Configuration Dialog Box

The Model Parameter Configuration dialog box is available only when the **Inline parameters** check box on the **Optimization** pane is selected. Selecting this check box activates the **Configure** button, as shown in the next figure.



Clicking the **Configure** button opens the Model Parameter Configuration dialog box.

Note The Model Parameter Configuration dialog box cannot tune parameters within referenced models. See “Parameterizing Model References” for tuning techniques that work with referenced models.



The Model Parameter Configuration Dialog Box

The Model Parameter Configuration dialog box lets you select base workspace variables and declare them to be tunable parameters in the current model. The dialog box is divided into two panels:

- The **Source list** panel displays a list of workspace variables and lets you add them to the tunable parameters list.
- The **Global (tunable) parameters** panel displays and maintains a list of tunable parameters associated with the model.

To declare tunable parameters, you select one or more variables from the **Source list**, add them to the **Global (tunable) parameters** list, and set their storage class and other attributes.

Source List Panel

The **Source list** panel displays a menu and a scrolling table of numerical workspace variables.

The menu lets you choose the source of the variables to be displayed in the list. There are two choices: **MATLAB workspace** (lists all variables in the MATLAB workspace that have numeric values), and **Referenced workspace variables** (lists only those variables referenced by the model). The source list displays names of variables defined in the MATLAB base workspace.

Selecting one or more variables from the source list enables the **Add to table** button. Clicking **Add to table** adds selected variables to the tunable parameters list in the **Global (tunable) parameters** panel. This action is all that is necessary to declare tunable parameters. However, if a block parameter which is not tunable is set to the name that appears on this list, a warning results during simulation and also during code generation.

In the **Source list**, the names of variables added to the tunable parameters list are displayed in bold type (see the preceding figure).

The **Refresh list** button updates the table of variables to reflect the current state of the workspace. If you define or remove variables in the workspace while the Model Parameter Configuration dialog box is open, click the **Refresh list** button when you return to the dialog box. The new variables are added to the source list.

Global (Tunable) Parameters Panel

The **Global (tunable) parameters** panel displays a scrolling table of variables that have been declared tunable in the current model and lets you specify their attributes. The **Global (tunable) parameters** panel also lets you remove entries from the list or create new tunable parameters.

You select individual variables and change their attributes directly in the table. The attributes are

- **Storage class** of the parameter in the generated code. Select one of
 - `SimulinkGlobal` (`Auto`)
 - `ExportedGlobal`
 - `ImportedExtern`
 - `ImportedExternPointer`

See “Tunable Parameter Storage Classes” on page 10-8 for definitions.

- **Storage type qualifier** of the variable in the generated code. For variables with any 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 additional qualifiers to the list. The code generator does not check the storage type qualifier for validity. The code generator includes the qualifier string in the generated code without syntax checking.
- **Name** of the parameter. This field is used only when creating a new tunable variable.

Use the **New** button to create a new tunable variable entry in the **Global (tunable) parameters** list. Enter the name and attributes of the variable and click **Apply**. The new variable does not need to be in use when you do this. At a later time, you can add references to any such variable in the model.

If the name you enter matches the name of an existing workspace variable in the **Source** list, that variable is declared tunable and appears in italics in the **Source** list.

Use the **Remove** button to delete selected variables from the **Global (tunable) parameters** list. All removed variables are inlined if **Inlined parameters** is on.

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).

Declaring Tunable Variables

To declare an existing variable tunable

- 1 Open the Model Parameter Configuration dialog box.
- 2 In the **Source** list panel, click the desired variable in the list to select it.
- 3 Click the **Add to table** button. The variable then appears in the table of tunable variables in the **Global (tunable) parameters** panel.

- 4** Click the variable in the **Global (tunable) parameters** panel.
- 5** Select the desired storage class from the **Storage class** menu.
- 6** Optionally, select (or enter) a storage type qualifier, such as `const` or `volatile` that you want the variable to have.
- 7** Click **Apply**, or click **OK** to apply changes and close the dialog box.

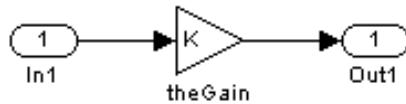
Tunable Expressions

The Real-Time Workshop product supports the use of tunable variables in expressions. An expression that contains one or more tunable parameters is called a *tunable expression*.

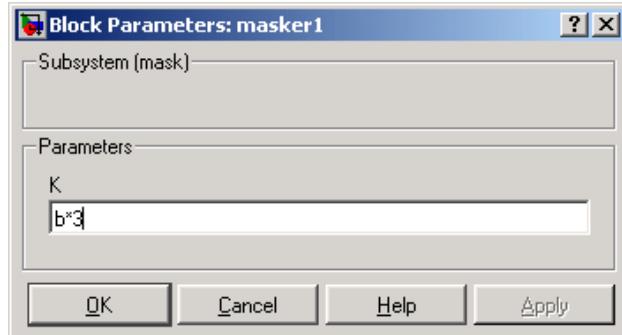
Tunable Expressions in Masked Subsystems

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 Real-Time Workshop 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 Real-Time Workshop 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 Real-Time Workshop 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” in the MATLAB reference documentation for more information on MATLAB typecasting.

- Category 5 operators are not supported.

Note The “dot” (structure membership) operator is not supported. This means that expressions that include a structure member are not tunable.

- 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 “Parameterizing 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 Real-Time Workshop 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

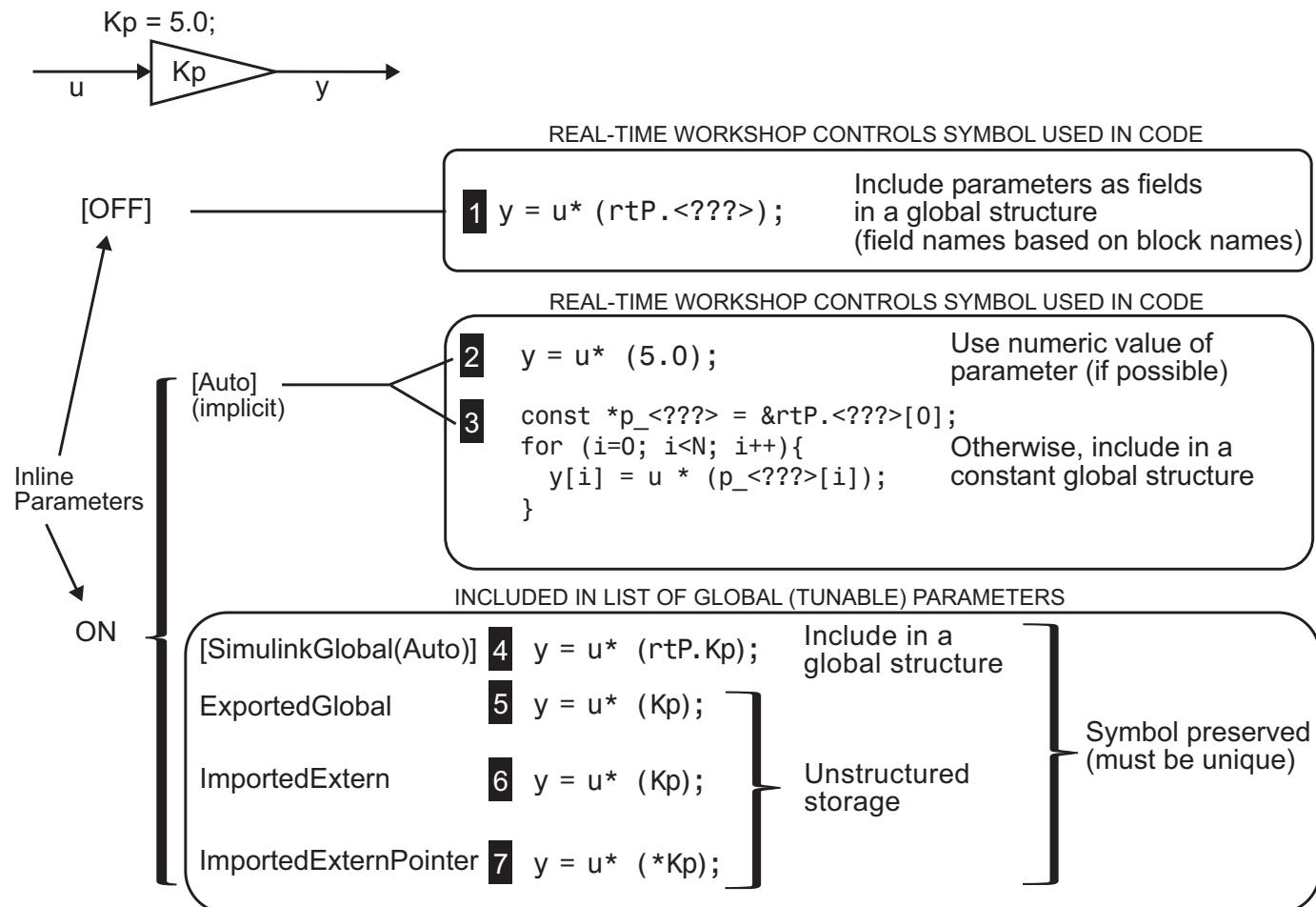
The Real-Time Workshop product can generate reusable (reentrant) code for a model containing identical atomic subsystems. Selecting the **Reusable** function option for **Real-Time Workshop system code** enables such code reuse, and causes a single function with arguments to be generated that is called when any of the identical atomic subsystem executes. See “Reusable

Function Option” on page 3-73 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 Real-Time Workshop product will generate separate functions for A and B.

Parameter Configuration Quick Reference Diagram

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



KEY:

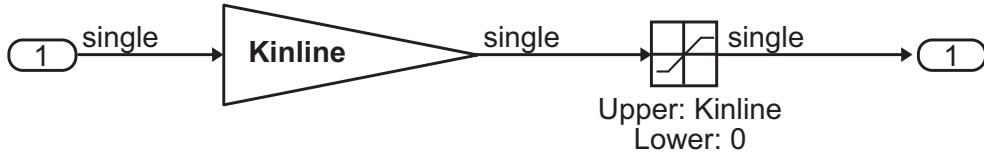
[option] : default for code generation option

<??> : RTW generated symbol for parameter storage field

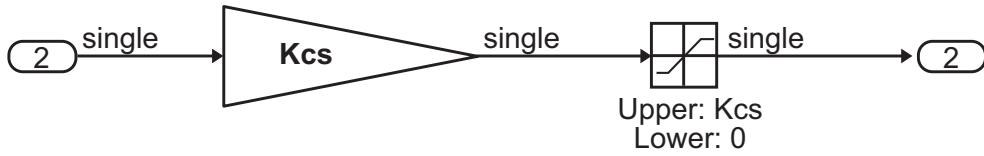
Generated Code for Parameter Data Types

For an example of the code generated from Simulink parameters with different data types, run the demo model `rtwdemo_paramdt`. This demo 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.

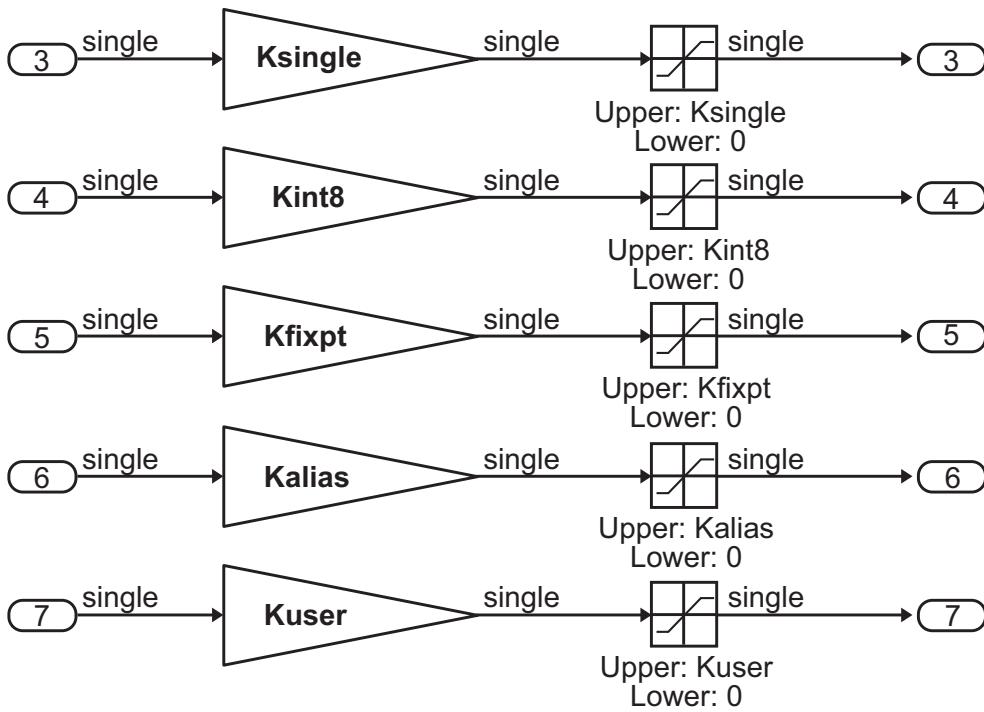
Inlined parameters (InLineParameters ON + Auto storage class)
 ==> numeric value inlined



Double-precision (context-sensitive) parameters
 ==> tunable parameter inherits data type from run-time parameter



Tunable parameters with explicit data type specification
 ==> parameter is cast to run-time parameter data type in generated code



The Simulink engine initializes the parameters in the demo 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 appropriate yellow boxes in the demo model.

In the demo model, note that the **Inline parameters** option on the **Optimization** pane of the **Configuration Parameters** dialog box is selected. The **Model Parameter Configuration** dialog box reveals that all 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 demo 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 <code>Kinline = 2;</code>	<pre>rtb_Gain1 = rtwdemo_paramdt_U.In1 * 2.0F; . . . rtwdemo_paramdt_Y.Out1 = rt_SATURATE(rtb_Gain1, 0.0F, 2.0F);</pre>
Kcs <code>Kcs = 3;</code>	<pre>real32_T Kcs = 3.0F; . . . rtb_Gain1 = rtwdemo_paramdt_U.In2 * Kcs; . . . rtwdemo_paramdt_Y.Out2 = rt_SATURATE(rtb_Gain1, 0.0F, Kcs);</pre>

10 Parameter Considerations

Parameter & MATLAB Code	Generated Variable Declaration and Code
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.RTWInfo.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))));

Parameter & MATLAB Code	Generated Variable Declaration and Code
Kalias <pre> aliasType = ... Simulink.AliasType('single'); Kalias = Simulink.Parameter; Kalias.Value = 7; Kalias.DataType = 'aliasType'; Kalias.RTWInfo.StorageClass = ... 'ExportedGlobal'; </pre>	<pre> 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); </pre>
Kuser <pre> userType = Simulink.NumericType; userType.DataMode = ... 'Fixed-point: slope and bias scaling'; userType.Slope = 2^-3; userType.isAlias = true; Kuser = Simulink.Parameter; Kuser.Value = 8; Kuser.DataType = 'userType'; Kuser.RTWInfo.StorageClass = ... 'ExportedGlobal'; </pre>	<pre> 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)))); </pre>

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

- The Real-Time Workshop 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 any transformation. Consequently, the Real-Time Workshop 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 appropriate 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 Real-Time Workshop product declares `Kalias` and `Kuser` to be of data types `aliasType` and `userType`, respectively.
- If a parameter specifies a fixed-point data type, the Real-Time Workshop product initializes its value in the generated code to the value of Q computed from the expression $V = SQ + B$ (see the Simulink Fixed Point 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** pane of the Configuration Parameters dialog box
- The parameter has a storage class other than Auto

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

If...	The Real-Time Workshop 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	<p>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:</p> <pre>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 these usages is permitted for any given variable.</pre>
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 (<code>int8</code> instead of <code>single</code>)	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 Real-Time Workshop product enforces limitations on the use of data types other than `double` in the workspace, as explained in “Limitations on Specifying Data Types in the Workspace Explicitly” on page 10-30.

Limitations on Specifying Data Types in the Workspace Explicitly

When you explicitly specify a data type other than `double` in the workspace, blocks typecast the parameter to the appropriate 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's Y data.)

Parameter Tuning by Using MATLAB Commands

When parameters are MATLAB workspace variables, the Model Parameter Configuration dialog box is the recommended way to see or set the attributes 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 “Configuring Parameter Objects for Code Generation” on page 12-4 for details.

The following commands return the tunable parameters and/or their attributes:

- `get_param(gcs, 'TunableVars')`
- `get_param(gcs, 'TunableVarsStorageClass')`
- `get_param(gcs, 'TunableVarsTypeQualifier')`

The following commands declare tunable parameters or set their attributes:

- `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 “Configuration Parameters for Simulink Models” in the Real-Time Workshop Reference.

Signal Considerations

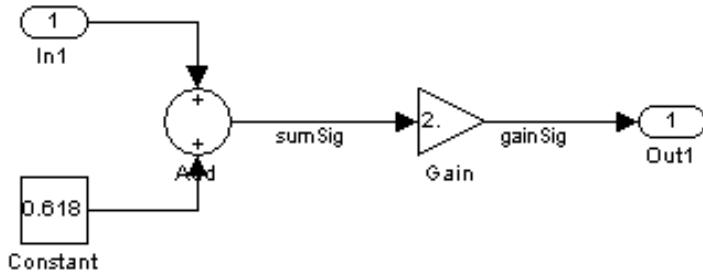
- “Introduction” on page 11-2
- “Signal Storage Concepts” on page 11-3
- “Signals with Auto Storage Class” on page 11-6
- “Signals with Test Points” on page 11-11
- “Interfacing Signals to External Code” on page 11-12
- “Symbolic Naming Conventions for Signals in Generated Code” on page 11-14
- “Summary of Signal Storage Class Options” on page 11-15
- “Monitoring Signals and Tuning Parameters” on page 11-17

Introduction

The Real-Time Workshop 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.
- Ensure that particular signals are stored in unique memory locations by declaring them as *test points*.
- Reduce memory usage by instructing the Real-Time Workshop 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 Real-Time Workshop product

The Global Block I/O Structure

By default, the Real-Time Workshop product attempts to optimize memory usage by sharing signal memory and using local variables.

However, there are a number of circumstances in which it is desirable or necessary to 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 (or all) 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 on page 11-2 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 in Generated Code” on page 11-14.

Signals Storage Classes

In the Real-Time Workshop 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 Real-Time Workshop 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. **Auto** is the appropriate storage class 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 11-6 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 appropriate 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:

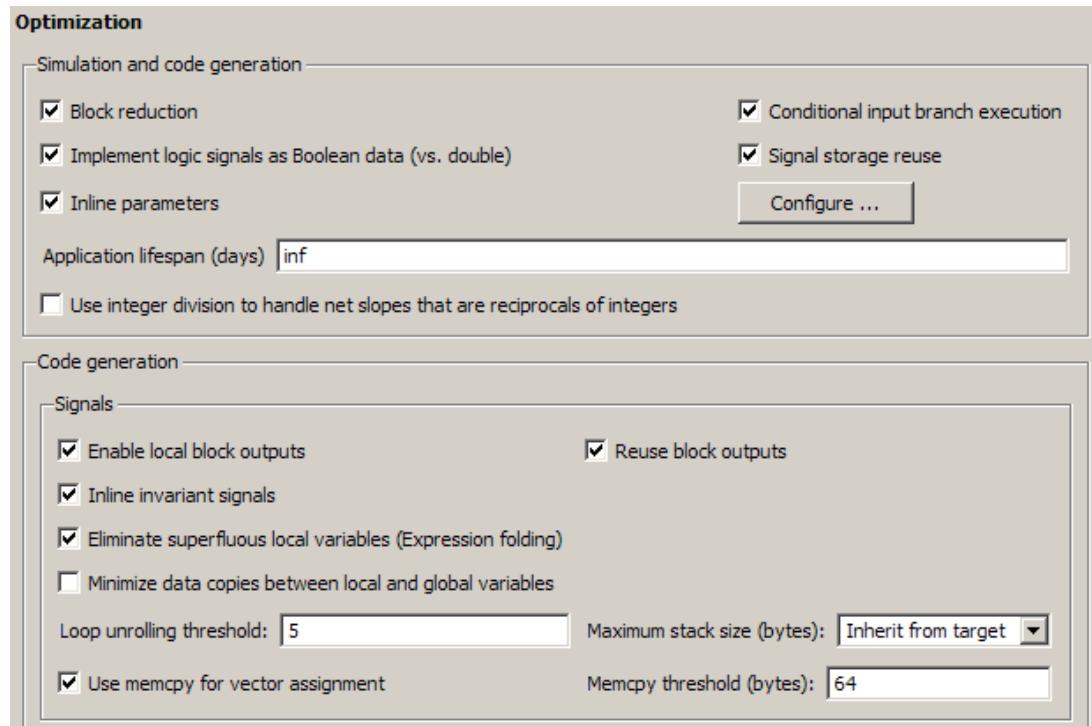
- **SimulinkGlobal(Test Point)**: Test points are stored as fields of the *model_B* structure that are not shared or reused by any other signal. See “Signals with Test Points” on page 11-11 for more information.
- **ExportedGlobal**: The signal is stored in a global variable, independent of the *model_B* data structure. *model.h* exports the variable. Signals with **ExportedGlobal** storage class must have unique signal names. See “Interfacing Signals to External Code” on page 11-12 for more information.
- **ImportedExtern**: *model_private.h* declares the signal as an **extern** variable. Your code must supply the proper variable definition. Signals with **ImportedExtern** storage class must have unique signal names. See “Interfacing Signals to External Code” on page 11-12 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 “Interfacing Signals to External Code” on page 11-12 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** pane of the Configuration Parameters dialog box, as shown in the next figure.



When you select **Signal storage reuse**, the **Enable local block outputs**, **Reuse block outputs**, **Eliminate superfluous local variables (Expression folding)**, and **Minimize data copies between local and global variables** options in the Code Generation section of the dialog box are enabled.

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 whenever possible.
- 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_exampl` model (see figure Signal_exampl Model on page 11-2).

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_exampl_Y.Out1`, a variable local to the model output function.

```
/* Model output function */
static void signal_exampl_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 <i>model_B</i> structure	Individual signal storage in <i>model_B</i> structure

Controlling Stack Space Allocation

The value of the “Maximum stack size (bytes)” parameter, on the Optimization pane of the Configuration parameter 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` appropriately 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 “Using Stack Space Allocation” on page 26-52 for more information.

Signals with Test Points

A *test point* is a signal that is stored in a unique location no other signals share or reuse. See “Working with 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 Real-Time Workshop 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 any 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 11-15.

If you have a Real-Time Workshop Embedded Coder license, you can specify that the Real-Time Workshop build process ignore all 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” in the *Real-Time Workshop Reference*.

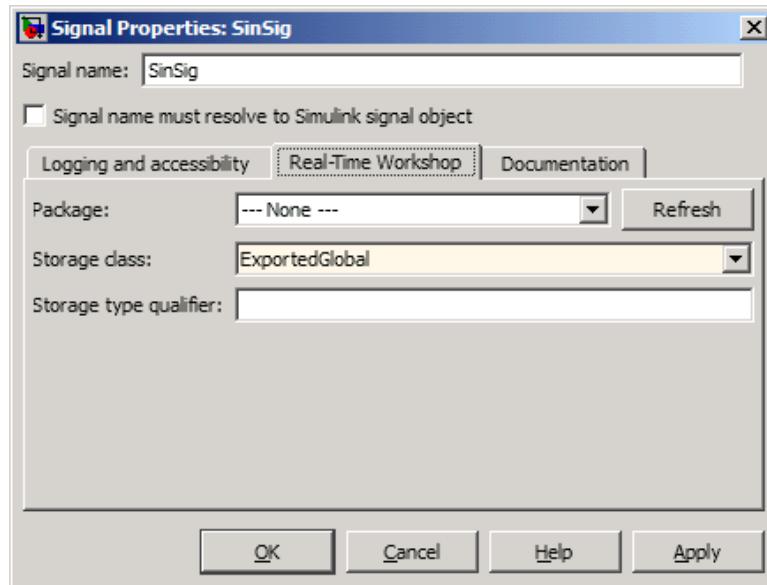
Interfacing Signals to External Code

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 **Real-Time Workshop** 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 **Real-Time Workshop** 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 Real-Time Workshop 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 Real-Time Workshop product provides C/C++ and Target Language Compiler APIs that give your code access to *model_B* and other data structures. See “Monitoring Signals and Tuning Parameters” on page 11-17 for more information.

Symbolic Naming Conventions for Signals in Generated Code

When signals have a storage class other than Auto, the Real-Time Workshop 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 on page 11-2).

```
/* 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 “Configuring Generated Identifiers” on page 20-3 for more detail.

When a signal has Auto storage class, the Real-Time Workshop 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 on page 11-2.

Storage Class	Declaration	Code
Auto (with signal storage reuse optimizations on)	In <code>model.h</code> <code>real_T rtb_sumSig;</code>	<pre>rtb_sumSig = signal_examp_U.In1 + signal_examp_P.Constant_Value; rtb_sumSig *= signal_examp_P.Gain_Gain; signal_examp_Y.Out1 = rtb_sumSig;</pre>
Test point (for <code>sumSig</code> only)	In <code>model.h</code> <code>typedef struct</code> <code>_BlockIO_signal_examp</code> <code>{</code> <code> real_T sumSig;</code> <code>}</code> <code>BlockIO_signal_examp;</code> In <code>model.c</code> or <code>model.cpp</code> <code>BlockIO_signal_examp</code> <code>signal_examp_B;</code> <code>real_T rtb_gainSig;</code>	<pre>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;</pre>
ExportedGlobal (for <code>sumSig</code> only)	In <code>model.h</code> <code>extern real_T sumSig;</code> In <code>model.c</code> or <code>model.cpp</code> <code>real_T sumSig;</code> <code>real_T rtb_gainSig;</code>	<pre>sumSig = signal_examp_U.In1 + signal_examp_P.Constant_Value; rtb_gainSig = sumSig * signal_examp_P.Gain_Gain; signal_examp_Y.Out1 = rtb_gainSig;</pre>

Storage Class	Declaration	Code
ImportedExtern	In <i>model_private.h</i> <pre>extern real_T sumSig;</pre> In <i>model.c</i> or <i>model.cpp</i> <pre>real_T rtb_gainSig;</pre>	<pre>sumSig = signal_examp_U.In1 + signal_examp_P.Constant_Value; rtb_gainSig = sumSig * signal_examp_P.Gain_Gain; signal_examp_Y.Out1 = rtb_gainSig;</pre>
ImportedExternPointer	In <i>model_private.h</i> <pre>extern real_T *sumSig;</pre> In <i>model.c</i> or <i>model.cpp</i> <pre>real_T rtb_gainSig;</pre>	<pre>(*sumSig) = signal_examp_U.In1 + signal_examp_P.Constant_Value; rtb_gainSig = (*sumSig) * signal_examp_P.Gain_Gain; signal_examp_Y.Out1 = rtb_gainSig;</pre>

Monitoring Signals and Tuning Parameters

In this section...

“Using the C API to Monitor Signals and States and Tune Parameters” on page 11-17

“Using the Target Language Compiler API to Monitor Signals and Tune Parameters” on page 11-17

Using the C API to Monitor Signals and States and Tune Parameters

The Real-Time Workshop product includes a C application program interface (API) for monitoring signals and states and tuning parameters independent of external mode. See Chapter 35, “Interacting with Target Application Data Using the C API” for information.

Using the Target Language Compiler API to Monitor Signals and Tune Parameters

The Real-Time Workshop product includes support for development of a Target Language Compiler API for monitoring signals and tuning parameters independent of external mode. See “Input Signal Functions”, “Output Signal Functions”, and “Parameter Functions” in the Target Language Compiler documentation for information.

Simulink Data Object Considerations

- “Introduction” on page 12-2
- “Parameter Objects” on page 12-4
- “Parameter Object Configuration Quick Reference Diagram” on page 12-11
- “Signal Objects” on page 12-12
- “Using Signal Objects to Initialize Signals and Discrete States” on page 12-18
- “Resolving Conflicts in Configuration of Parameter and Signal Objects” on page 12-26
- “Customizing Code for Parameter and Signal Objects” on page 12-29
- “Using Objects to Export ASAP2 Files” on page 12-29

Introduction

Before using Simulink data objects with the Real-Time Workshop product, read the following:

- The discussion of Simulink data objects in the Simulink documentation
- Chapter 10, “Parameter Considerations”
- Chapter 11, “Signal Considerations”

Within the class hierarchy of Simulink data objects, the Simulink product provides two classes that are designed as base classes for signal and parameter storage:

- **Simulink.Parameter**: Objects that are instances of the **Simulink.Parameter** class or any class derived from **Simulink.Parameter** are called *parameter objects*.
- **Simulink.Signal**: Objects that are instances of the **Simulink.Signal** class or any class derived from **Simulink.Signal** are called *signal objects*.

The **RTWInfo** properties of parameter and signal objects are used by the Real-Time Workshop product during code generation. These properties let you assign storage classes to the objects, thereby controlling how the generated code stores and represents signals and parameters.

The Real-Time Workshop build process also writes information about the properties of parameter and 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 the Target Language Compiler documentation.

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

- 1 Define a subclass of one of the built-in **Simulink.Data** classes.
 - For parameters, define a subclass of **Simulink.Parameter**.
 - For signals, define a subclass of **Simulink.Signal**.

-
- 2** Instantiate parameter or signal objects from your subclass and set their properties appropriately, from the command line or using Model Explorer.
 - 3** Use the objects as parameters or signals within your model.
 - 4** Generate code and build your target executable.

The following sections describe the relationship between Simulink data objects and code generation in the Real-Time Workshop product.

Parameter Objects

This section discusses how to use parameter objects in code generation.

Configuring 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 Chapter 10, “Parameter Considerations”).
- 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 any Simulink numeric data type, including a fixed-point, user-defined, or alias data type.
 - **RTWInfo.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.

Note If **Inline parameters** is off (the default), the `RTWInfo.StorageClass` parameter object property is ignored in code generation.

Effect of Storage Classes on Code Generation for Parameter Objects

The Real-Time Workshop product generates code and storage declarations based on the `RTWInfo.StorageClass` property of the parameter object. The logic is as follows:

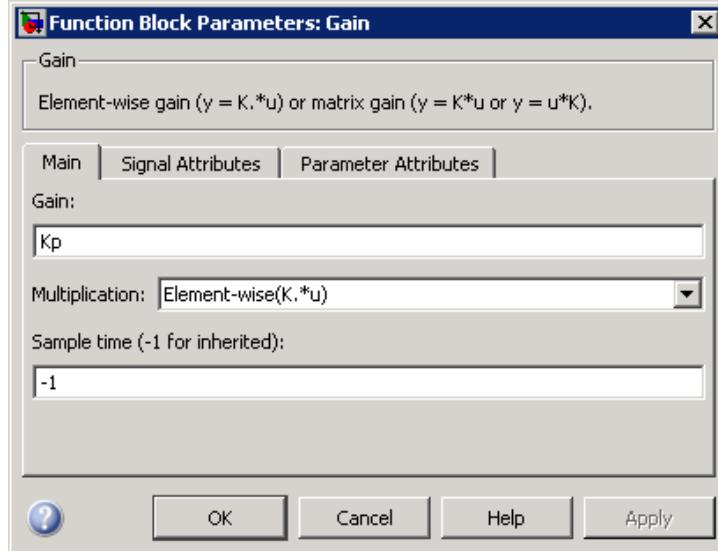
- If the storage class is 'Auto' (the default), the parameter object is inlined (if possible), using the `Value` property.
- For storage classes other than 'Auto', the parameter object is handled as a tunable parameter.

- 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 “Controlling Parameter Object Code Generation Using the Model Explorer” on page 12-7 for examples of code generated for possible settings of `RTWInfo.StorageClass`.

Controlling Parameter Object Code Generation with Typed Commands

In this section, the Gain block computations of the model shown in the next figure are used as an example of how the Real-Time Workshop 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
    RTWInfo: [1x1 Simulink.ParamRTWInfo]
    Description: ''
    DataType: 'auto'
    Min: -Inf
    Max: Inf
    DocUnits: ''
    Complexity: 'real'
    Dimensions: '[1x1]'
```

Make sure that the name of the parameter object matches the desired block parameter in your model. This ensures that the Simulink engine can associate the parameter name with the correct 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 `RTWInfo.StorageClass` property, for example:

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

The `RTWInfo` parameters are now

```
Kp.RTWInfo
Simulink.ParamRTWInfo
    StorageClass: 'ExportedGlobal'
    Alias: ''
    CustomStorageClass: 'Default'
```

```
CustomAttributes: [1x1
SimulinkCSC.AttribClass_Simulink_Default]
```

Controlling Parameter Object Code Generation Using the Model Explorer

If you prefer, you can create and modify attributes of parameter objects using the Model Explorer. This lets you see all 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:

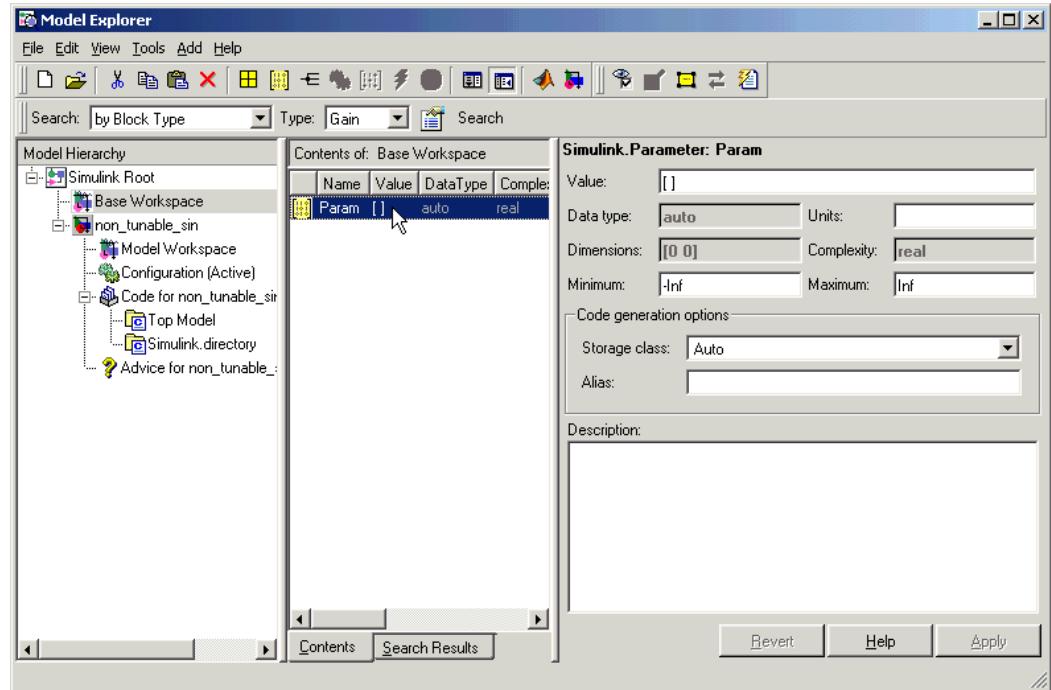
- 1 Choose **Model Explorer** from the **View** menu.

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:

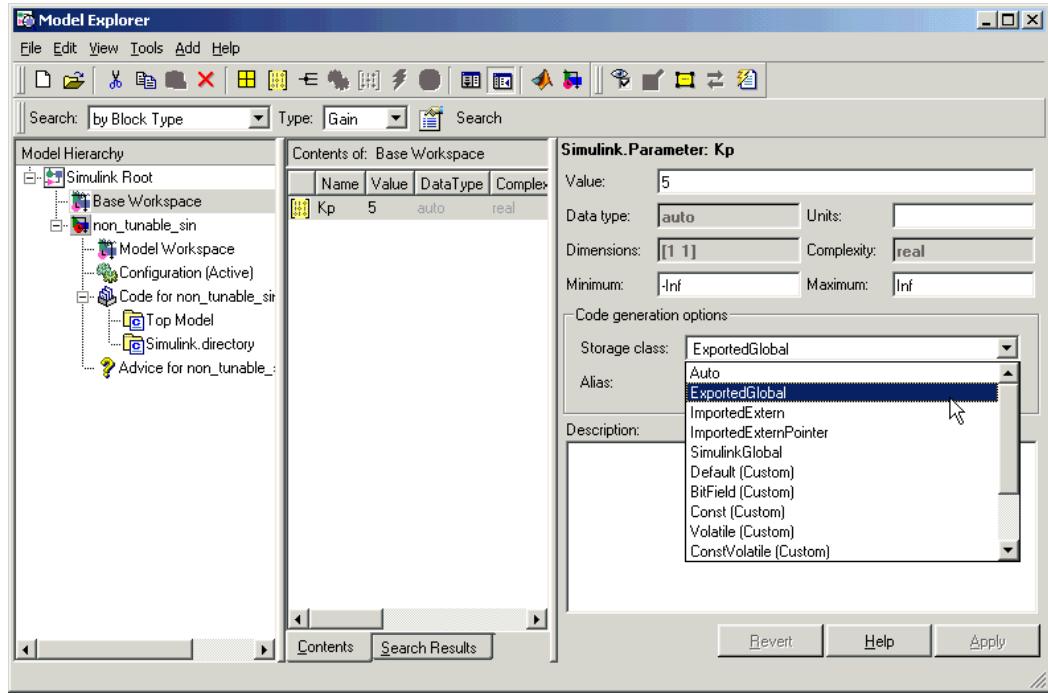
- a Click the word Param in the Name column to select it.
- b Rename it by typing Kp in place of Param.
- c Click Return.

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.RTWInfo.StorageClass in Model Explorer:

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



- b Click Apply.

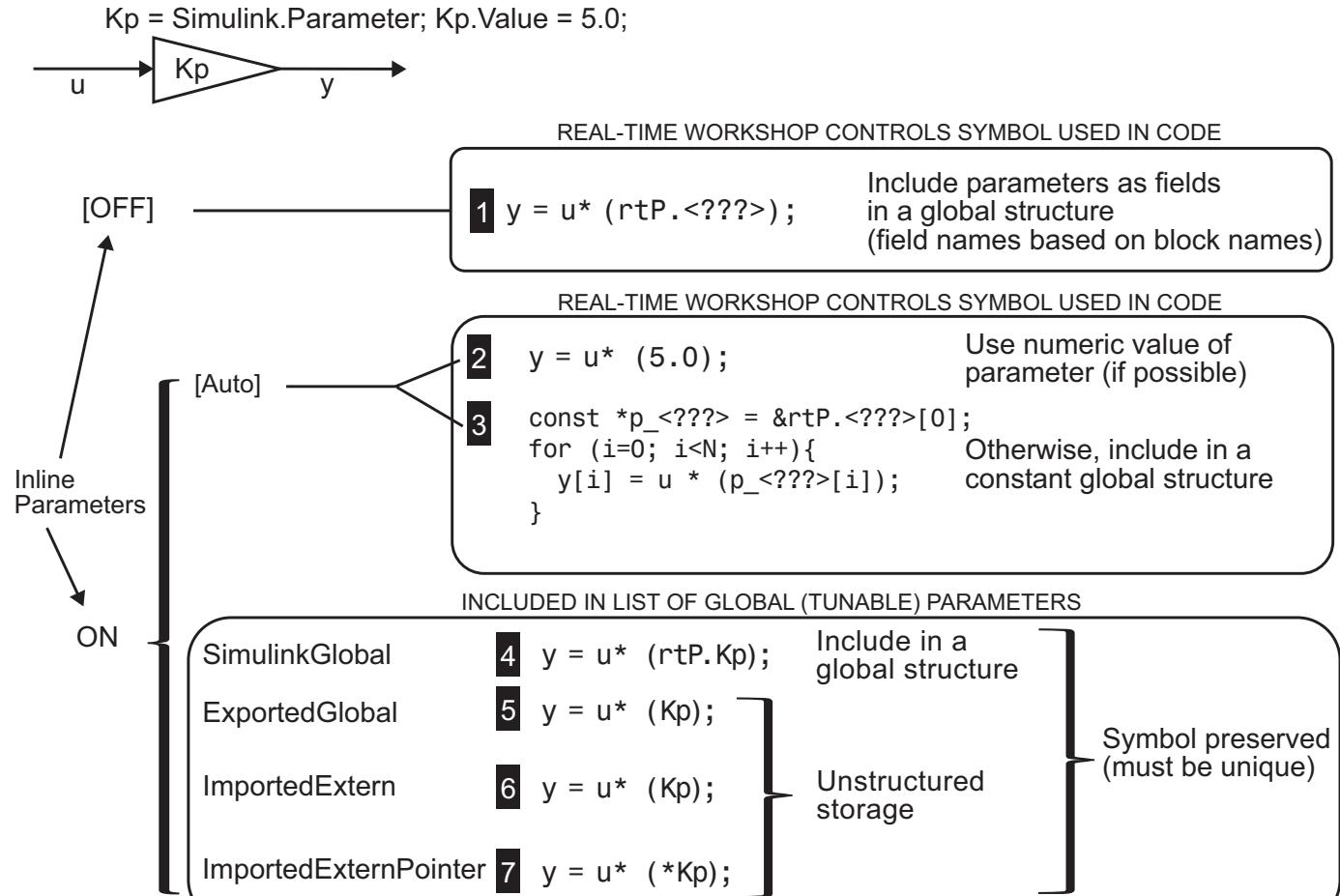
The following table shows the variable declarations for Kp and the code generated for the Gain block in the model shown in the preceding model, with the **Inline parameters** and **Eliminate superfluous local variables (Expression folding)** check boxes selected (which includes the gain computation in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

StorageClass Property	Generated Variable Declaration and Code
Auto	<pre><i>model_Y</i>.Out1 = rtb_u * 5.0;</pre>
SimulinkGlobal	<pre>struct _Parameters_<i>model</i> { real_T Kp; } . Parameters_<i>model</i> <i>model_P</i> = { 5.0 }; . <i>model_Y</i>.Out1 = rtb_u * <i>model_P</i>.Kp;</pre>
ExportedGlobal	<pre>extern real_T Kp; . real_T Kp = 5.0; . <i>model_Y</i>.Out1 = rtb_u * Kp;</pre>

StorageClass Property	Generated Variable Declaration and Code
ImportedExtern	<pre>extern real_T Kp; . . . <i>model_Y</i>.Out1 = rtb_u * Kp;</pre>
ImportedExternPointer	<pre>extern real_T *Kp; . . . <i>model_Y</i>.Out1 = rtb_u * (*Kp);</pre>

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.


KEY:

[option] : default for code generation option

<????> : RTW generated symbol for parameter storage field

Signal Objects

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 12-4.

Configuring 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 Chapter 11, “Signal Considerations”).
- The **Enable local block outputs** code generation option (see Chapter 11, “Signal Considerations”).
- The **Minimize data copies between local and global variables** code generation option (see Chapter 11, “Signal Considerations”).
- The **RTWInfo.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 11-4).

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

Effect of Storage Classes on Code Generation for Signal Objects

The way in which the Real-Time Workshop 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 `RTWInfo.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 Real-Time Workshop 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**

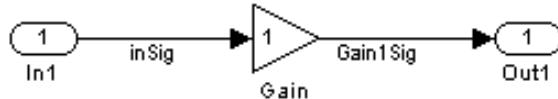
global variables code generation options to determine whether signal objects are stored in reusable and/or local variables. Make sure that these options are set correctly for your application.

To generate a test point or signal storage declaration that can interface externally, use an explicit `RTWInfo.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.RTWInfo.StorageClass = 'SimulinkGlobal';
```

Controlling Signal Object Code Generation By Using Typed Commands

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,

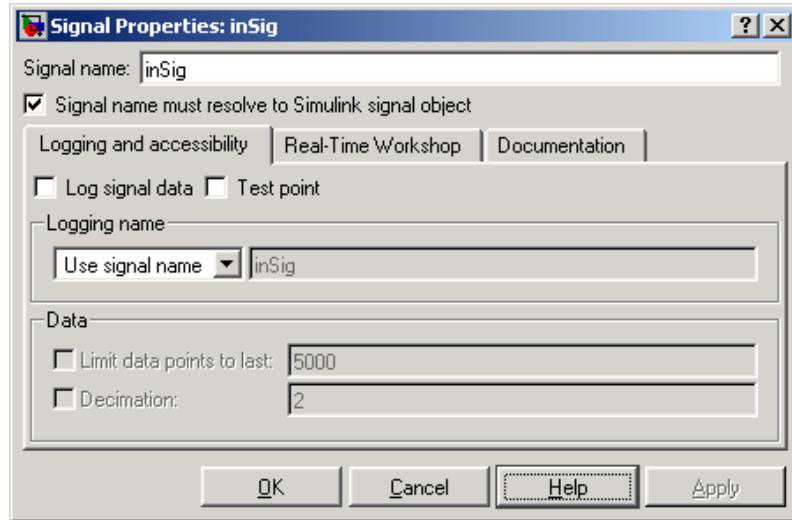
- 1 Define a subclass of `Simulink.Signal`. In this example, the signal object is an instance of the class `Simulink.Signal`, which is provided with the Simulink product.
- 2 Instantiate a signal object from your subclass. The following example instantiates `inSig`, a signal object of class `Simulink.Signal`.

```
inSig = Simulink.Signal
inSig =
Simulink.Signal
    RTWInfo: [1x1 Simulink.SignalRTWInfo]
    Description: ''
    DataType: 'auto'
    Min: -Inf
    Max: Inf
    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 ensures that the Simulink engine can resolve the signal label to the correct 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 `RTWInfo.StorageClass` property as follows.

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

Controlling Signal Object Code Generation By 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:

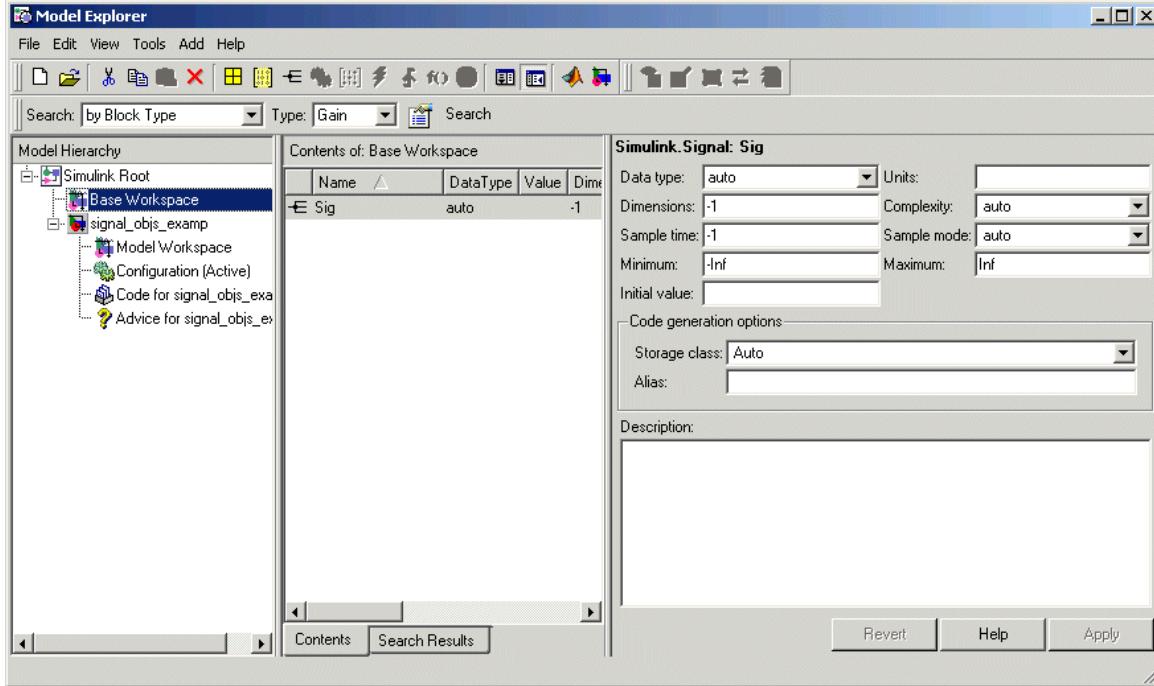
- 1 Choose **Model Explorer** from the View menu.

Model Explorer opens or activates if it already was open.

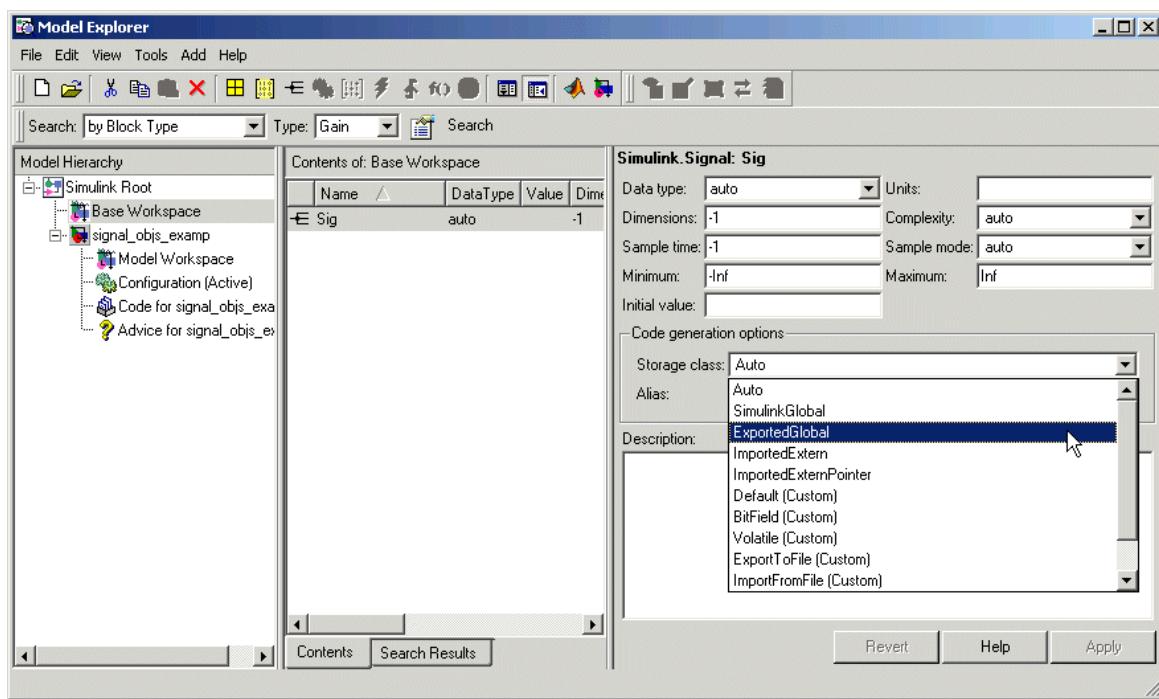
- 2 Select Base Workspace in the **Model Hierarchy** pane.

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

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

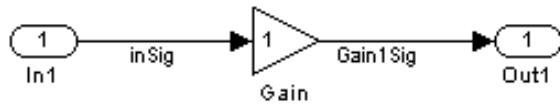


- 4 To set the signal name in Model Explorer, click the word **Sig** in the **Name** column to select it, and rename it by typing **inSig** followed by **Return** in place of **Sig**.
- 5 To set the **inSig.RTWInfo.StorageClass** in Model Explorer, click the **Storage class** menu and select **ExportedGlobal**, as shown in the next figure.



6 Click **Apply**.

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



Storage Class	Declaration	Code
Auto (with storage optimizations on)	In <i>model.h</i> <pre>typedef struct _ExternalInputs_signal_ objs_examp_tag { real_T inSig; } ExternalInputs_signal_ objs_examp;</pre>	rtb_Gain1Sig = signal_objs_examp_U.inSig * signal_objs_examp_P.Gain_Gain;
Simulink Global	In <i>model.h</i> <pre>typedef struct _ExternalInputs_signal_objs_examp_tag { real_T inSig; } ExternalInputs_signal_objs_examp;</pre>	rtb_Gain1Sig = signal_objs_examp_U.inSig * signal_objs_examp_P.Gain_Gain;
ExportedGlobal	In <i>model.c</i> or <i>model.cpp</i> <pre>real_T inSig;</pre> In <i>model.h</i> <pre>extern real_T inSig;</pre>	rtb_Gain1Sig = inSig * signal_objs_examp_P.Gain_Gain;
ImportedExtern	In <i>model_private.h</i> <pre>extern real_T inSig;</pre>	rtb_Gain1Sig = inSig * signal_objs_examp_P.Gain_Gain;
ImportedExternPoint	In <i>model_private.h</i> <pre>extern real_T *inSig;</pre>	rtb_Gain1Sig = (*inSig) * signal_objs_examp_P.Gain_Gain;

Using Signal Objects to Initialize Signals and Discrete States

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.

Specifying an Initial Value for a 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
- Any 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.

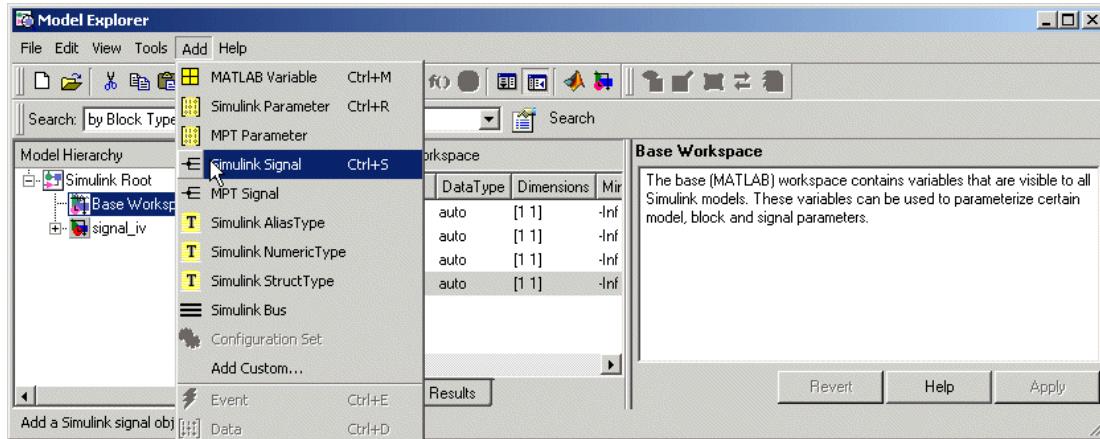
Note Initial value settings for signal objects that represent the following signals and states override the corresponding block parameter initial values if undefined (specified as []):

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

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

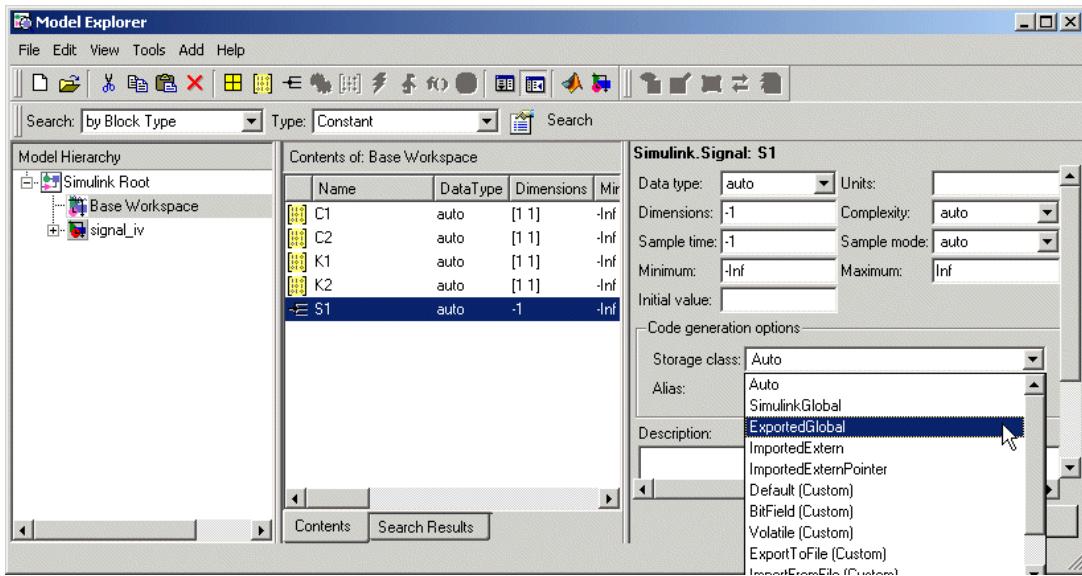
```
S1=Simulink.Signal;
```

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 ensures consistency between signal objects in the MATLAB workspace and the 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

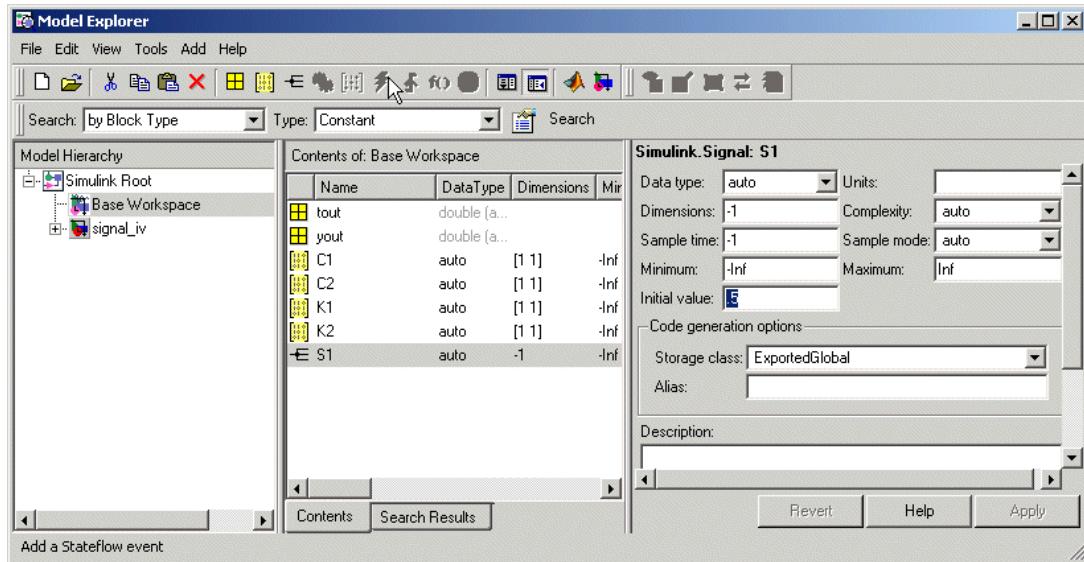
```
S1.RTWInfo.StorageClass='ExportedGlobal';
```

- 3** Set the initial value. You can specify any 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';

If necessary, the Simulink engine converts the initial value to ensure type, complexity, and dimension consistency with the corresponding block parameter value. If you specify an invalid value or expression, an error message appears when you update the model.

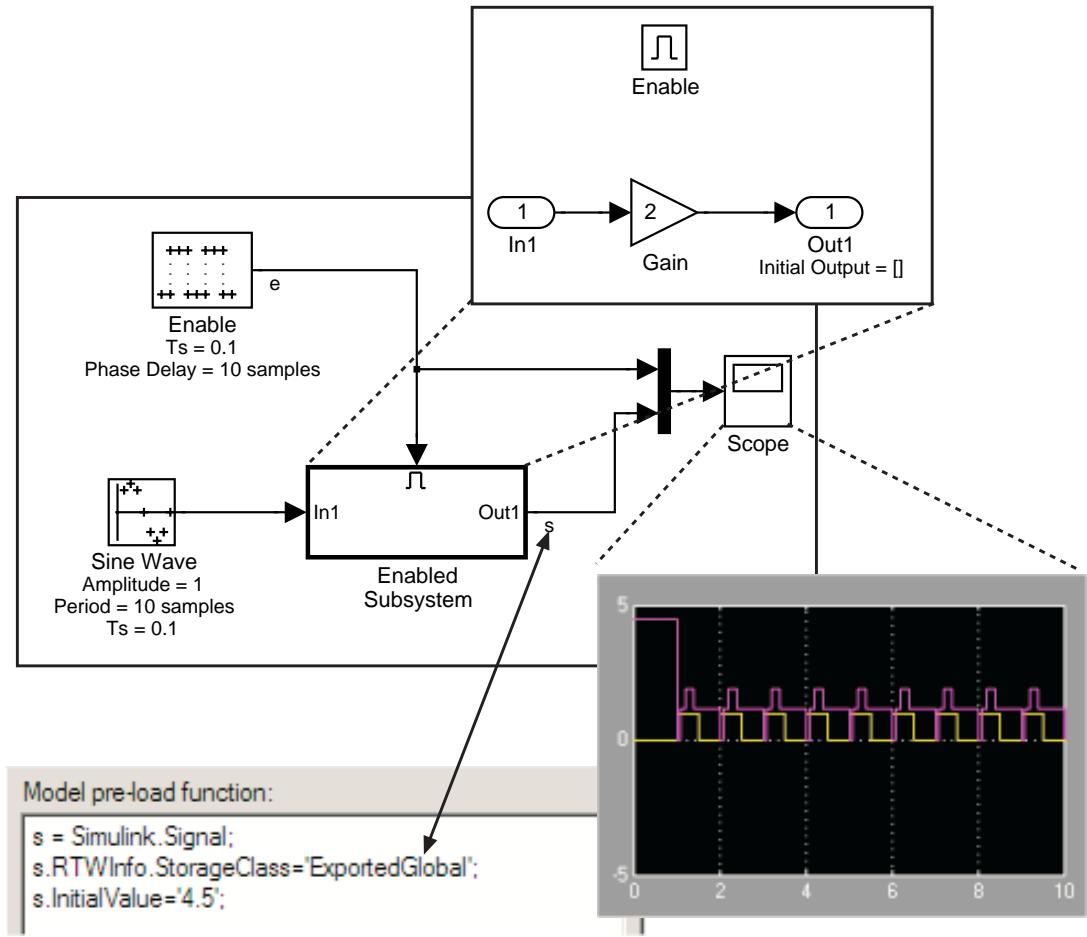
Model Explorer



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:

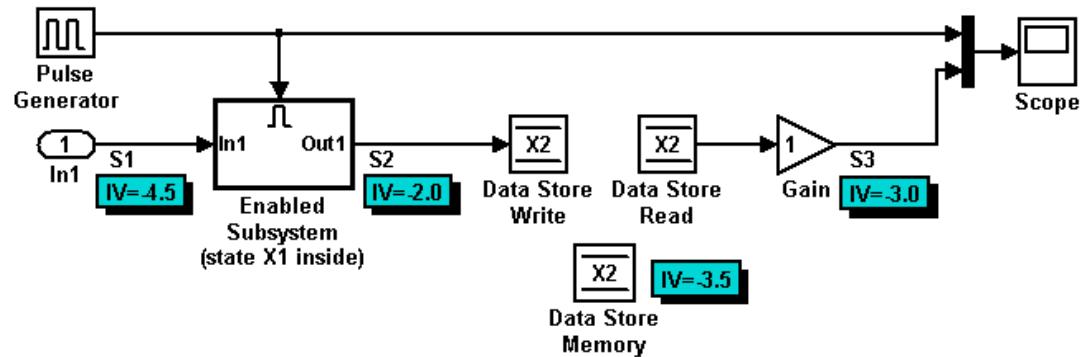
- RSim executables can use the **Data Import/Export** pane of the Configuration Parameters dialog box to load input values from MAT-files. GRT and ERT executables cannot load input values from MAT-files.

- 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 demo model *rtwdemo_sigobj_iv*.



If you create and initialize signal objects in the base workspace, the Real-Time Workshop 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(boolean_T firstTime)

{
    .
    .
    .
}
```

```
/* 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 Chapter 10, “Parameter Considerations”.

Resolving Conflicts in Configuration of Parameter and Signal Objects

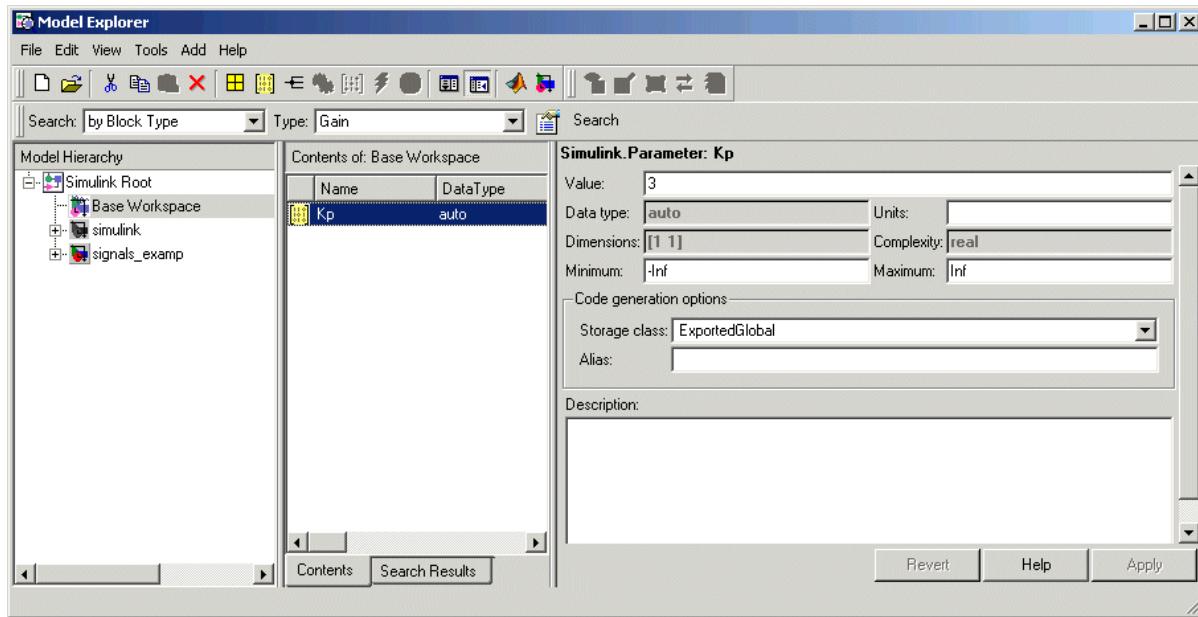
This section describes how to avoid and resolve certain conflicts that can arise when using parameter and signal objects.

Parameters

As explained in Chapter 12, “Simulink Data Object Considerations” and “Using the Model Parameter Configuration Dialog Box” on page 10-11, two methods are available for controlling the tunability of parameters. You can

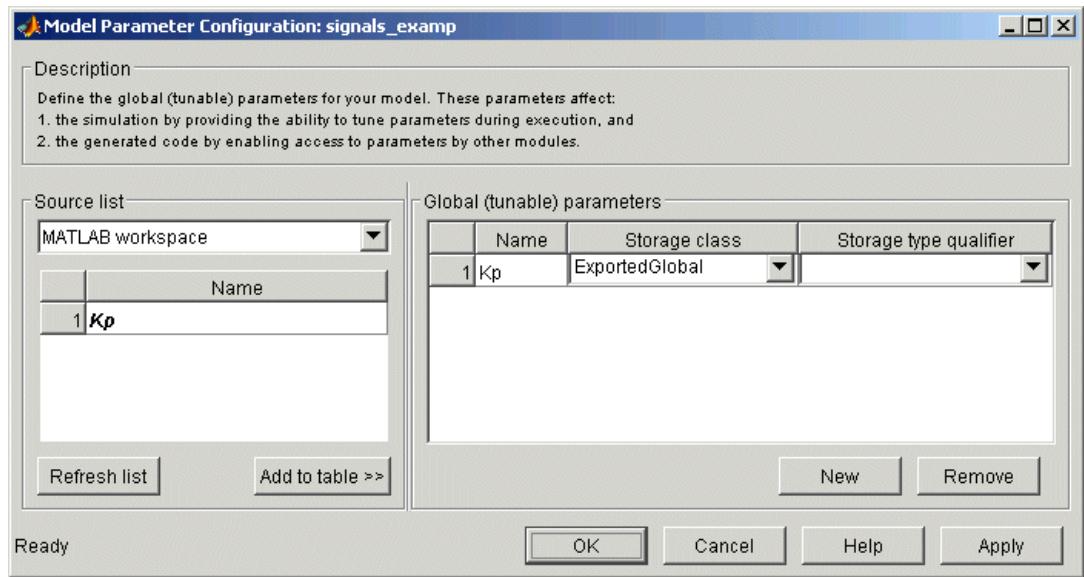
- Define them as `Simulink.Parameter` objects in the MATLAB workspace
- Use the Model Parameter Configuration dialog box

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.

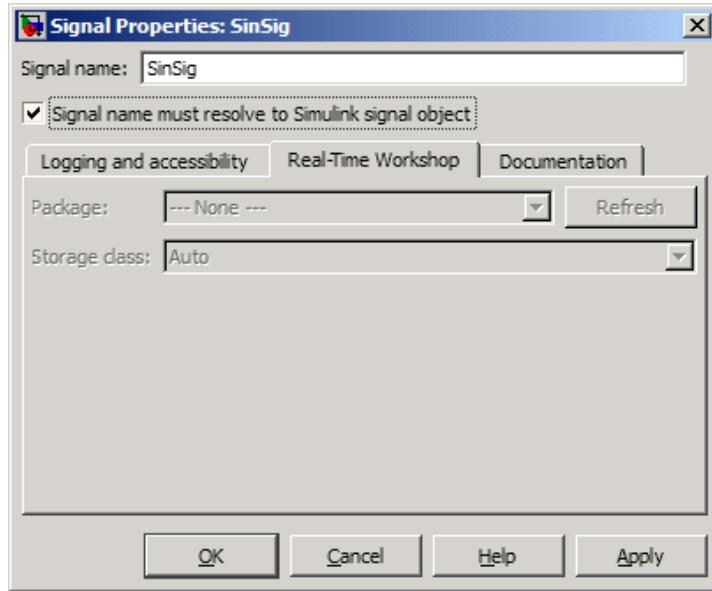


Parameter Kp Defined with Simulink Global Storage Class

Note The MathWorks™ recommends that you 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 and Block States

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 no longer can specify the **Storage class** property in the **Real-Time Workshop** pane 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, no conflict arises, and `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.

Customizing Code for Parameter and Signal Objects

You can influence the treatment of parameter and signal objects in generated code by using TLC to access fields in object records in *model.rtw* files. For details on doing this, see the Target Language Compiler documentation.

Using Objects to Export ASAP2 Files

The Real-Time Workshop product provides an interface for exporting ASAP2 files, which you customize. For details, see Chapter 36, “Generating Model Information for Host-Based ASAP2 Data Measurement and Calibration”.

Enumerated Data Type Considerations

- “About Enumerated Data Types” on page 13-2
- “Default Code for an Enumerated Data Type” on page 13-3
- “Enumerated Type Safe Casting” on page 13-4
- “Overriding Default Methods (Optional)” on page 13-5
- “How emlc Works with Enumerated Types” on page 13-9
- “Enumerated Type Limitations” on page 13-10

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 all 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 “Using Enumerated Data” in the Simulink documentation. For information about enumerated data types in Stateflow charts, see “Using Enumerated Data in Stateflow Charts”.

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
```

Enumerated Type Safe Casting

When code generated for a Simulink Data Type Conversion block or a Stateflow block casts data to an enumerated type, and the block's **Saturate on integer overflow** option is selected, the cast uses a safe-cast function. The following code shows a safe-cast function for `BasicColors`:

```
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;
}
```

The cast fails if the value to be cast does not correspond to one of the enumerated values in the enumerated type. When a safe cast fails, the value returned is the underlying integer of the enumerated type's default value. The above code reflects this default for `BasicColors`. See "Specifying a Default Enumerated Value" in the Simulink documentation for more information.

When a block's **Saturate on integer overflow** option is cleared, and the block casts to an enumerated type, the resulting code does not use safe casting. The code is therefore more efficient, but is more vulnerable to runtime errors. No warning about the lack of safe casting appears during code generation.

Overriding Default Methods (Optional)

Every enumerated class has four associated static methods, which it inherits from `Simulink.IntEnumType`. You can optionally override any or all of 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 any of the methods, include a customized version of the method in the enumerated class definition’s `methods` section. If you do not want to override any 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
<code>getDefaultValue</code>	Returns the default value for the class, which must be an instance of the class.	The lexically first value in the enumeration.	Any enumerated value in the class. See “Instantiating an Enumerated Type”.
<code>getDescription</code>	Returns a string containing a description of the enumerated class.	''	Any string that MATLAB accepts.

Method	Purpose	Default Return	Custom Return
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 any legal MATLAB string for *description*. The generated code that defines the enumerated type will include the specified description.

Specifying 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 where needed in the code.
% Otherwise, the type is written out in the generated code.
    retVal = 'filename';
end
```

Substitute any 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.

Prefixing 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()
% ADDCLASSTNAMETOENUMNAMES 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 13-2, 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.

How emlc Works with Enumerated Types

The `emlc` command generates code for enumerated data types supported by the Embedded MATLAB subset. The subset supports two integer-based enumerated types:

- Enumerated types based on `Simulink.IntEnumType` (available with a Simulink license)

Use enumerated types based on `Simulink.IntEnumType` with `emlc` when generating C executables or library code from Embedded MATLAB-compliant MATLAB code. For information about defining enumerated types based on `Simulink.IntEnumType`, see “Defining an Enumerated Data Type” in the Simulink documentation.

`Simulink.IntEnumType` provides four static methods that you can override in your type definition to customize the behavior of enumerated data for simulation and code generation. To learn about these methods, see “Overriding Default Methods (Optional)” on page 13-5.

- Enumerated types based on `int32`

Use enumerated types based on `int32` with `emlc` when generating C-MEX functions from Embedded MATLAB-compliant MATLAB code, for rapid prototyping and verification of generated C code within MATLAB.

You can also use this enumerated type when generating C code with `emlc`, but `int32` does not provide methods for customizing the behavior of enumerated data. For information about defining enumerated types based on `int32`, see “Using Enumerated Data in Embedded MATLAB Compliant Code” in the Embedded MATLAB documentation.

Enumerated Type Limitations

- Generated code does not support logging enumerated data.

Block State Storage and Interfacing Considerations

- “Introduction” on page 14-2
- “Block State Storage” on page 14-2
- “Block State Storage Classes” on page 14-4
- “Using the State Attributes Tab to Interface States to External Code” on page 14-6
- “Symbolic Names for Block States” on page 14-8
- “Block States and Simulink Signal Objects” on page 14-11
- “Summary of State Storage Class Options” on page 14-12

Introduction

For certain block types, the Real-Time Workshop product lets you control how block states in your model are stored and represented in the generated code. Using the **State Attributes** tab of a block dialog box, you can:

- Control whether or not 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.
- Assign symbolic names to block states in generated code.

Block State Storage

The discussion of block state storage in this section applies to the following blocks:

- Discrete Filter
- 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. For more information on the DWork vector, see the Target Language Compiler documentation.

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.

Block State Storage Classes

The storage class property of a block state specifies how the Real-Time Workshop 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. Auto is the appropriate storage class 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 Real-Time Workshop product generates one, as described in “Symbolic Names for Block States” on page 14-8.

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 appropriate 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 proper 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 proper pointer variable

definition. States with `ImportedExternPointer` storage class must have unique names.

The table in “Summary of Signal Storage Class Options” on page 11-15 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 Real-Time Workshop product generates one, as described in “Symbolic Names for Block States” on page 14-8.

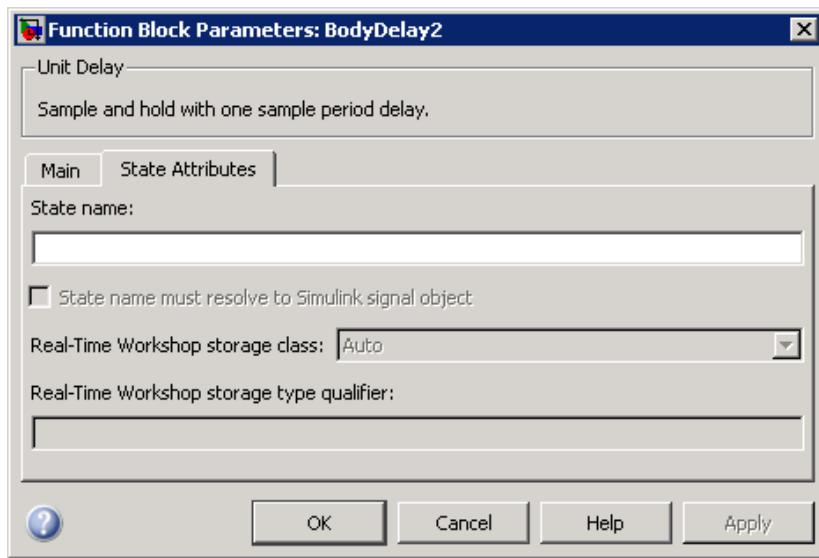
The next section explains how to use the **State Attributes** tab of the block dialog box to assign storage classes to block states.

Using the State Attributes Tab to 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 **Real-Time Workshop 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 **Real-Time Workshop 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 **Real-Time Workshop storage type qualifier** field. The Real-Time Workshop 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 Block States

To determine the variable or field name generated for a block's state, you can:

- Use a default name generated by the Real-Time Workshop 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

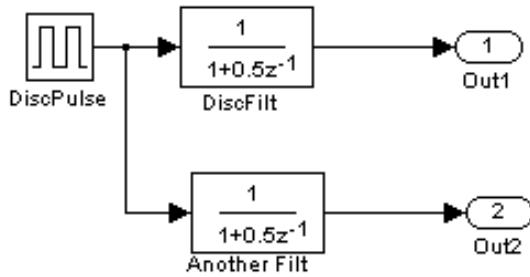
If you do not define a symbolic name for a block state, the Real-Time Workshop product uses the following default naming convention:

`BlockType#_DSTATE`

where

- `BlockType` is the name of the block type (for example, `Discrete_Filter`).
- `#` is a unique ID number (#) assigned by the Real-Time Workshop 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 always 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:

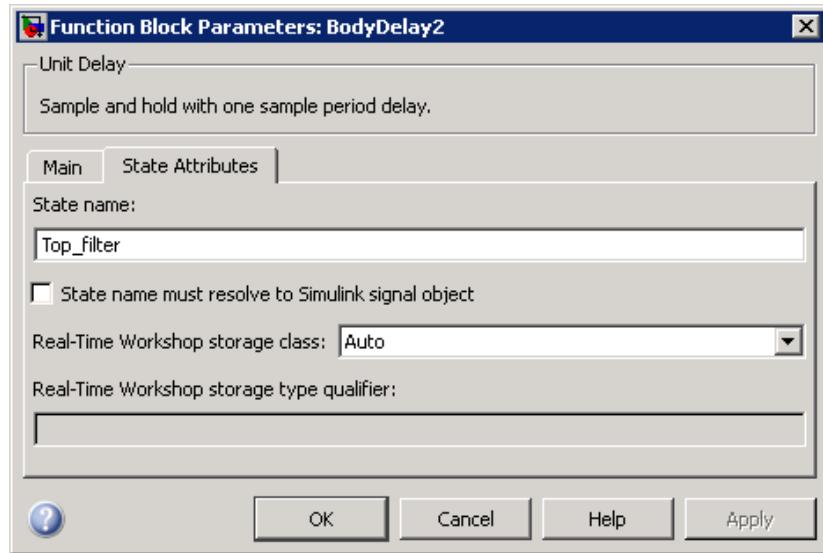
```
/* data type work */
disc_filt_states_M->Work.dwork = ((void *)
&disc_filt_states_DWork);
(void)memset((char_T *) &disc_filt_states_DWork, 0,
sizeof(D_Work_disc_filt_states));
{
    int_T i;
    real_T *dwork_ptr = (real_T *)
&disc_filt_states_DWork.DiscFilt_DSTATE;

    for (i = 0; i < 2; i++) {
        dwork_ptr[i] = 0.0;
    }
}
```

User-Defined 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 “Controlling Signal Object Code Generation By Using Typed Commands” on page 12-13, 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;
```

Block States and Simulink Signal Objects

If you are not familiar with Simulink data objects and signal objects, you should read Chapter 12, “Simulink Data Object Considerations” 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 `RTWInfo.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**.

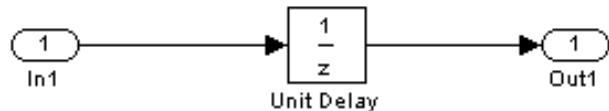
This action disables the **Real-Time Workshop storage class** and **Real-Time Workshop storage type qualifier** options in the **State Attributes** tab, because the signal object specifies these settings.

- 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.mdl`, 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 <code>model.h</code> <pre> typedef struct D_Work_unit_delay_tag { real_T udx; } D_Work_unit_delay;</pre>	<code>unit_delay_DWork.udx = 0.0;</code>
Exported Global	In <code>model.c</code> or <code>model.cpp</code> <pre> real_T udx;</pre> In <code>model.h</code> <pre> extern real_T udx;</pre>	In <code>model.c</code> or <code>model.cpp</code> <pre> udx = 0.0;</pre>

Storage Class	Declaration	Initialization Code
ImportedExtern	In <i>model_private.h</i> extern real_T udx;	In <i>model.c</i> or <i>model.cpp</i> udx = unit_delay_P.UnitDelay_X0;
ImportedExternPointer	In <i>model_private.h</i> extern real_T *udx;	In <i>model.c</i> or <i>model.cpp</i> (*udx) = unit_delay_P.UnitDelay_X0;

Data Store Memory Considerations

- “About Simulink Data Stores” on page 15-2
- “Storage Classes for Data Store Memory Blocks” on page 15-3
- “Data Store Memory Blocks and Simulink Signal Objects” on page 15-6
- “Nonscalar Data Stores in Generated Code” on page 15-7
- “Data Store Buffering in Generated Code” on page 15-10

About Simulink Data Stores

A data store contains data that is accessible at any point 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 “Working with Data Stores” 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.

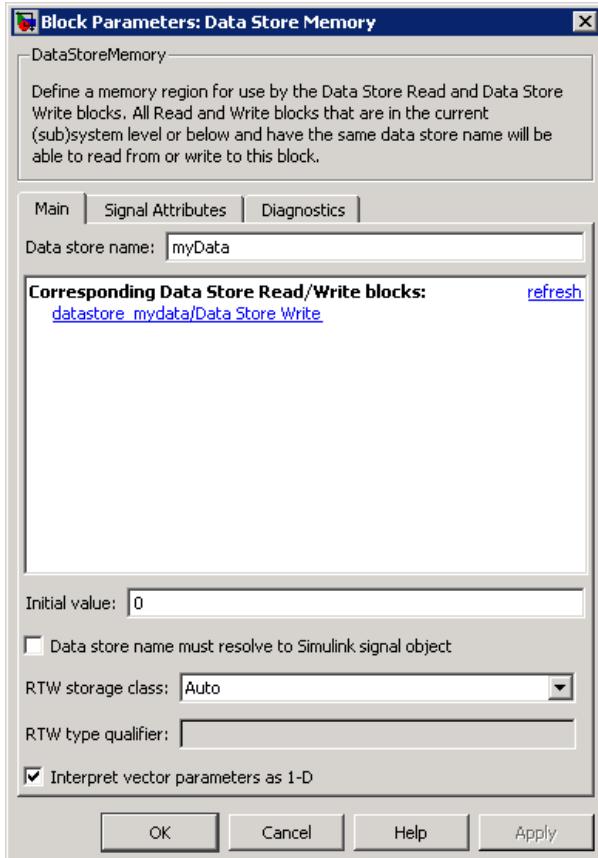
You can generate code from multiple Data Store Memory blocks that have the same data store name, subject to the following restriction: *at most one* of the identically named blocks can have a storage class other than `Auto`. An error is reported if this condition is not met.

For blocks with `Auto` storage class, the Real-Time Workshop product generates a unique symbolic name for each block (if necessary) 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 **RTW storage class** and **RTW 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 **RTW storage class**. The table gives the variable declarations and **MdlOutputs** code generated for the **myData** block.

Storage Class	Declaration	Code
Auto	<p>In <i>model.h</i></p> <pre>typedef struct D_Work_tag { real_T myData; } D_Work;</pre> <p>In <i>model.c</i> or <i>model.cpp</i></p> <pre>/* Block states (auto storage) */ D_Work model_DWork;</pre>	<pre>model_DWork.myData = rtb_SineWave;</pre>
ExportedGlobal	<p>In <i>model.c</i> or <i>model.h</i></p> <pre>/* Exported block states */ real_T myData;</pre> <p>In <i>model.h</i></p> <pre>extern real_T myData;</pre>	<pre>myData = rtb_SineWave;</pre>
ImportedExtern	<p>In <i>model_private.h</i></p> <pre>extern real_T myData;</pre>	<pre>myData = rtb_SineWave;</pre>
ImportedExternPointer	<p>In <i>model_private.h</i></p> <pre>extern real_T *myData;</pre>	<pre>(*myData) = rtb_SineWave;</pre>

Data Store Memory Blocks and Simulink Signal Objects

If you are not familiar with Simulink data objects and signal objects, you should read Chapter 12, “Simulink Data Object Considerations” 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:

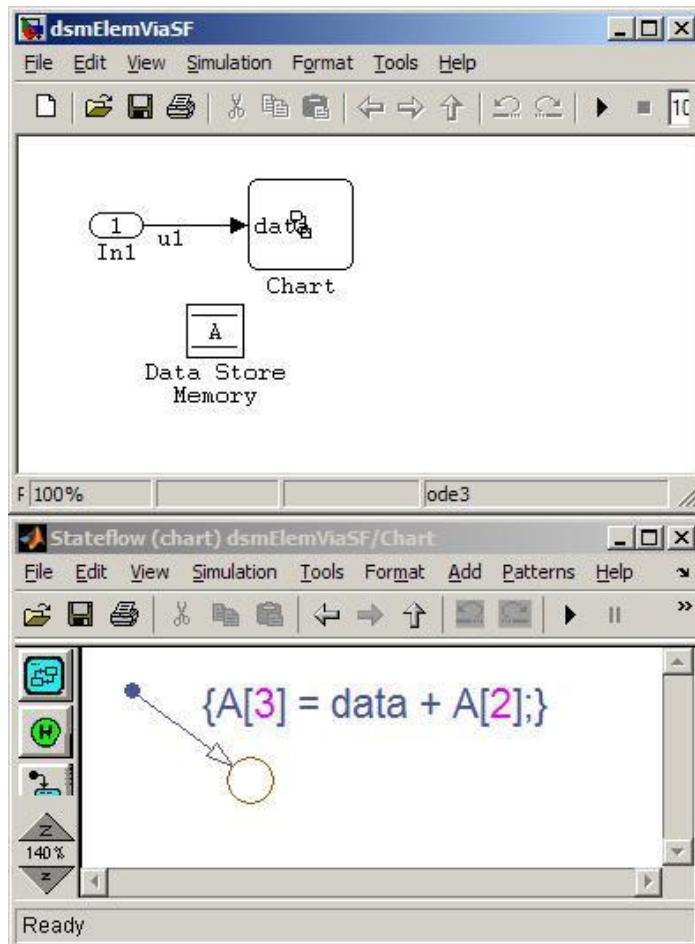
- 1** Instantiate the desired signal object.
- 2** Set the object's `RTWInfo.StorageClass` property to indicate the desired storage class.
- 3** Open the block dialog box for the Data Store Memory block that you want to associate with the signal object.
- 4** Enter the name of the signal object in the **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 “Resolving Conflicts in Configuration of Parameter and Signal Objects” on page 12-26 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 is efficient and involves no unnecessary access of any of the other elements of A.

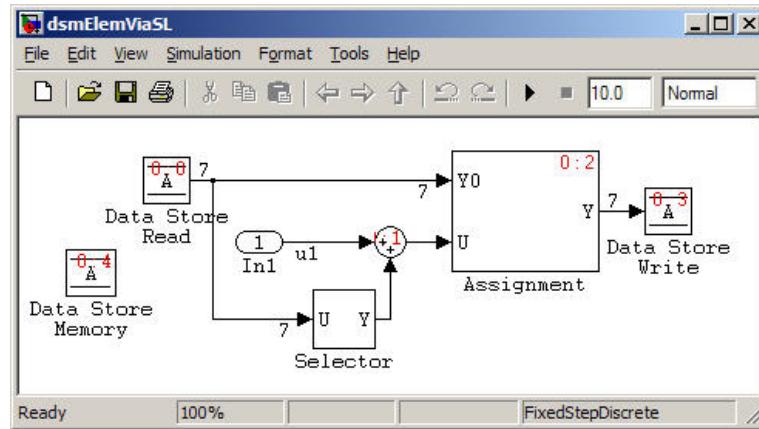
15 Data Store Memory Considerations



```
/* 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

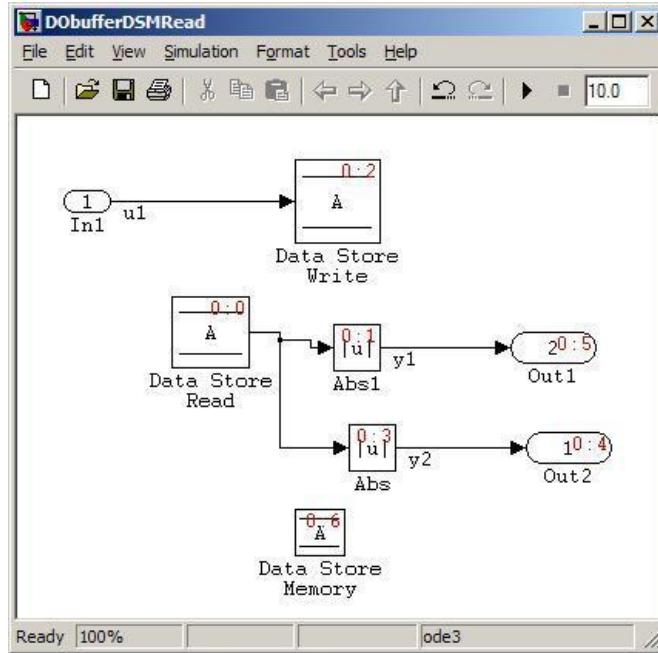
copies each element of the data store back to itself, in addition to updating the element.



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, all downstream blocks connected to the output of the data store read utilize the same value, even if a Data Store Write block updates the data store in between execution of two of the downstream blocks.

The next figure shows a model that uses blocks whose priorities have been modified to achieve a particular order of execution:



```

/* local block i/o variables */
real_T rtb_DataStoreRead;

/* DataStoreRead: '<Root>/Data Store Read' */
rtb_DataStoreRead = A; /* Buffer the value of A */

/* Abs: '<Root>/Abs1' incorporates:
 * DataStoreRead: '<Root>/Data Store Read'
 */
y1 = fabs(A); /* Use A (whose value equals
the buffered value at this point)

/* DataStoreWrite: '<Root>/Data Store Write' incorporates:
 * Inport: '<Root>/In1'
 */
A = u1; /* Update the value of A

/* Abs: '<Root>/Abs' */
y2 = fabs(rtb_DataStoreRead); /* Consistently use the same buffered
value as before the update to A

```

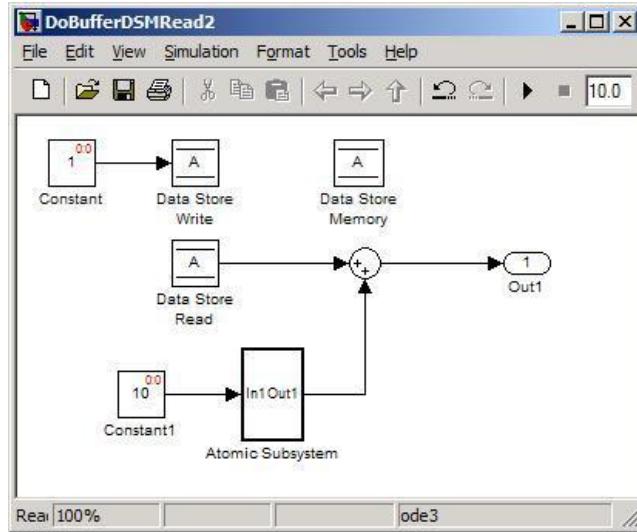
The following execution order applies:

- 1 The block Data Store Read buffers the current value of the data store A at its output.

- 2** The block Abs1 uses the buffered output of Data Store Read.
- 3** The block Data Store Write updates the data store.
- 4** The block Abs uses the buffered output of Data Store Read.

Because the output of Data Store Read is a buffer, both Abs and Abs1 use the same value: the value of the data store at the time that Data Store Read executes.

The next figure shows another example:



```

real_T rtb_DataStoreRead;

/* DataStoreWrite: '<Root>/Data Store Write' incorporates:
 * Constant: '<Root>/Constant'
 */
A = DoBufferDSMRead2_P.Constant_Value;

/* DataStoreRead: '<Root>/Data Store Read' */
rtb_DataStoreRead = A; Buffer the value of A

/* Outputs for atomic SubSystem: '<Root>/Atomic Subsystem' */
DoBufferDSMRead_AtomicSubsystem(); We don't do a global analysis to detect if this function writes to A

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

/* Outport: '<Root>/Out1' incorporates:
 * Sum: '<Root>/Sum'
 */
DoBufferDSMRead2_Y.Out1 = rtb_DataStoreRead + DoBufferDSMRead2_B.Abs;
Use the buffered value of A

```

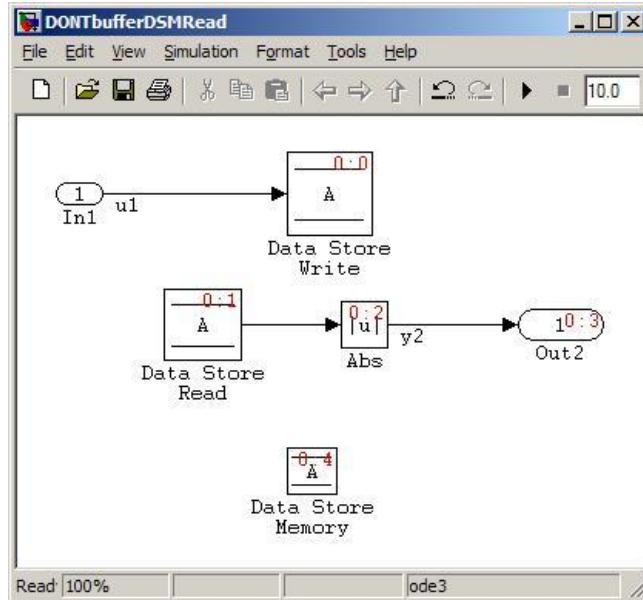
In this example, the following execution order applies:

- 1** The block Data Store Read buffers the current value of the data store A at its output.
- 2** Atomic Subsystem executes.

- 3 The Sum block adds the output of Atomic Subsystem to the output of Data Store Read.

Simulink assumes that Atomic Subsystem might update the data store, so Simulink buffers the data store to ensure correct results. Atomic Subsystem executes after Data Store Read buffers its output, and the buffer guarantees that the Sum block will use the value of the data store as it was when Data Store Read executed.

In some cases, the Real-Time Workshop code generator determines that it can safely 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:
     *   Import: '<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.

Preparing Models for Code Generation

- Chapter 16, “Configuring a Model for Code Generation”
- Chapter 17, “Enabling Instrumentation for Debugging”
- Chapter 18, “Selecting and Configuring a Target ”
- Chapter 19, “Selecting a Target Programming Language”
- Chapter 20, “Specifying Code Appearance and Documentation”
- Chapter 21, “Customizing Targets”



Configuring a Model for Code Generation

- “Getting Familiar With Model Configuration Options That Pertain To Code Generation” on page 16-2
- “Configuring Code Generation Objectives” on page 16-4
- “Configuring a Model from the MATLAB Command Window” on page 16-5

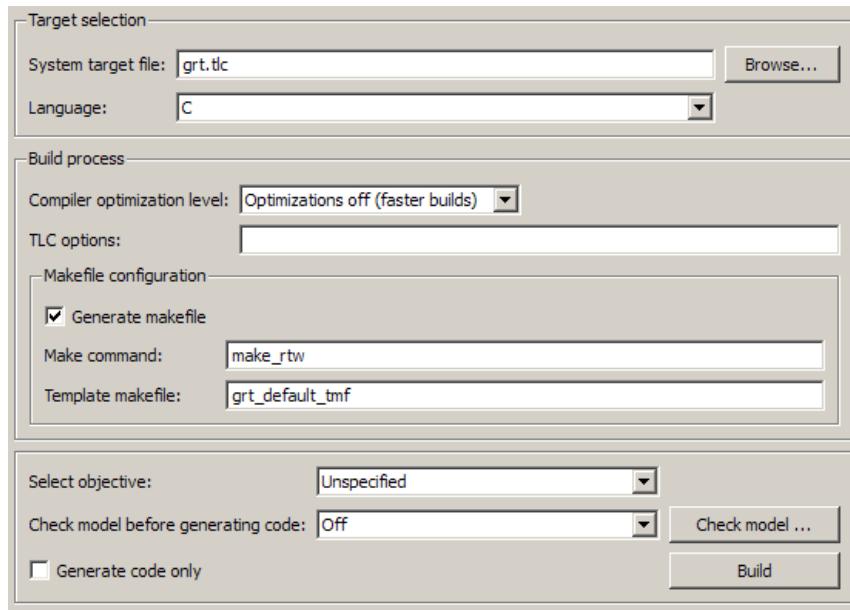
Getting Familiar With Model Configuration Options That Pertain To Code Generation

Opening the Real-Time Workshop Pane

There are three ways to open the **Real-Time Workshop** pane of the Configuration Parameters dialog box:

- From the **Simulation** menu, choose **Configuration Parameters**. When the Configuration Parameters dialog box opens, click **Real-Time Workshop** in the **Select** (left) pane.
- Select **Model Explorer** from the **View** menu in the model window, or type `daexplr` on the MATLAB command line and press **Enter**. In the Model Explorer, expand the node for the current model in the left pane and click **Configuration (active)**. The configuration dialog elements are listed in the middle pane. Clicking any of these brings up that dialog in the right pane. Alternatively, right-clicking the **Real-Time Workshop** configuration element in the middle pane and choosing **Properties** from the context menu activates that dialog in a separate window.
- Select **Options** from the **Real-Time Workshop** submenu of the **Tools** menu in the model window.

The general **Real-Time Workshop** pane, as it appears in the Model Explorer, appears in the next figure.



Real-Time Workshop Pane

This pane allows you to specify most of the options for controlling the Real-Time Workshop code generation and build process. The content of the pane and its subpanes can change depending on the target you specify. Thus, a model that has multiple configuration sets can invoke parameters in one configuration that do not apply to another configuration. In addition, some configuration options are available only with the Real-Time Workshop Embedded Coder product.

For descriptions of **Real-Time Workshop** pane parameters, see “Real-Time Workshop Pane: General” in the Real-Time Workshop reference.

Configuring Code Generation Objectives

You can specify high-level code generation objectives for your model and run Code Generation Advisor checks. The software then identifies changes to model constructs and settings that can improve the generated code. You can make the changes using the Code Generation Advisor interface.

The high-level code generation objectives are:

- Execution efficiency (ERT-based targets)
- ROM efficiency (ERT-based targets)
- RAM efficiency (ERT-based targets)
- Safety precaution (ERT-based targets)
- Traceability (ERT-based targets)
- Debugging (all targets)

To specify code generation objectives, choose a system target file and then specify the objectives using the **Select objectives** menu (GRT-based targets) or the **Set objectives** button (ERT-based targets).

After specifying your code generation objectives, you can run Code Generation Advisor checks before generating code by clicking the **Check model** button. You can also run the Code Generation Advisor during code generation by setting the **Check model before generating code** to one of the **On** options.

Running the Code Generation Advisor identifies suggested changes to your model that you can make using the Code Generation Advisor interface. The generated code includes comments identifying which high-level objectives you specified, and the results of the Code Generation Advisor run at the time of code generation.

For detailed information, see “Mapping Application Objectives to Model Configuration Parameters” in the Real-Time Workshop Embedded Coder documentation.

Configuring a Model from the MATLAB Command Window

When you are ready to generate code for a model, you should consider adjusting the model's simulation configuration parameters to values that are optimal for code generation. One way of adjusting the parameters is to modify option settings in the Configuration Parameters dialog box. Alternatively, you can use the `set_param` function. The user interface options and associated parameters related to the Real-Time Workshop and Real-Time Workshop Embedded Coder products are described in “Configuration Parameters for Simulink Models” in the Real-Time Workshop reference. This section describes simulation parameter adjustments to consider for code generation.

Note When you change a check box, menu selection, or edit field in any Configuration Parameters dialog box, the white background of the element you altered turns to light yellow to indicate that an unsaved change has been made. When you click **OK**, **Cancel**, or **Apply**, the background resets to white.

Many model configuration parameters affect the way that the Real-Time Workshop software generates code and builds an executable from your model.

However, you initiate and directly control the code generation and build process from the **Real-Time Workshop** pane and related tabs (also presented as subnodes).

In addition to using the Configuration Parameters dialog box, you can use `get_param` and `set_param` to individually access most configuration parameters. The configuration parameters you can get and set are listed in “Parameter Command-Line Information Summary” in the Real-Time Workshop reference.

You can use the Model Advisor to help configure any model to optimally achieve your code generation objectives. See “Getting Advice About Optimizing Models for Code Generation” on page 26-5 for more information.

Enabling Instrumentation for Debugging

Use parameters on the **Diagnostics** and **Real-Time Workshop > Debug** panes of the Configuration Parameter dialog box to configure a model such that generated code and the build process are optimized for troubleshooting. You can set parameters that apply to the model compilation phase, the target language code generation phase, or both.

Parameters in the following table will be helpful if you are writing TLC code for customizing targets, integrating legacy code, or developing new blocks.

To...	Select...
Display progress information during code generation in the MATLAB Command Window	Verbose build. Compiler output also displays.
Prevent the build process from deleting the <i>model.rtw</i> file from the build directory at the end of the build	Retain .rtw file. This parameter is useful if you are modifying the target files, in which case you need to look at the <i>model.rtw</i> file.

To...	Select...
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 Real-Time Workshop 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 Real-Time Workshop code generator reads each line of TLC code during code generation	Start TLC coverage when generating code. Alternatively, enter the argument -dg for the System Target File parameter on the Real-Time Workshop pane.
Halt a build if any 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 <code>set_param</code> command. The default is off. <code>set_param(model, 'TLCAssertion', 'on off')</code> To check the current setting, use <code>get_param</code> . <code>get_param(model, 'TLCAssertion')</code>

To...	Select...
Detect loss of tunability	<p>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:</p> <ul style="list-style-type: none"> • none — Loss of tunability can occur without notification. • warning — Loss of tunability generates a warning (default). • error — Loss of tunability generates an error. <p>For a list of supported operators and functions, see “Tunable Expression Limitations” on page 10-18</p>
Enable model verification (assertion) blocks	<p>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 has the same affect on generated code as it does on simulation behavior. For example, simulation and code generation ignore this parameter when model verification blocks are inside an S-function. Possible values are:</p> <ul style="list-style-type: none"> • User local settings • Enable All • Disable All <p>For Assertion blocks not disabled, generated code for a model includes one of the following statements, at appropriate locations, depending on the blocks input signal type (Boolean, real, or integer, respectively).</p>

To...	Select...
	<pre>utAssert(input_signal); utAssert(input_signal != 0.0); utAssert(input_signal != 0);</pre> <p>By default, <code>utAssert</code> has no effect in generated code. For assertions to abort execution, you must enable them by specifying the following <code>make_rtw</code> command for Real-Time Workshop > Make command:</p> <pre>make_rtw OPTS="-DDOASSERTS"</pre> <p>Use the following variant if you want triggered assertions to print the assertion statement instead of aborting execution:</p> <pre>make_rtw OPTS="-DDOASSERTS -DPRINT_ASSERTS"</pre> <p><code>utAssert</code> is defined as <code>#define utAssert(exp) assert(exp).</code></p> <p>To customize assertion behavior, provide your own definition of <code>utAssert</code> in a handwritten header file that overrides the default <code>utAssert.h</code>. For details on how to include a customized header file in generated code, see Chapter 30, “Integrating External Code Using Model Configuration Parameters”.</p> <p>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.</p>

See the Target Language Compiler documentation for details. Also, consider using the Model Advisor as a tool for troubleshooting model builds.

For descriptions of Debug pane parameters, see “Real-Time Workshop Pane: Debug” in the Real-Time Workshop reference documentation.

Selecting and Configuring a Target

- “About Targets and Code Formats” on page 18-2
- “Targets and Code Formats” on page 18-3
- “Targets and Code Styles” on page 18-4
- “Backwards Compatibility of Code Formats” on page 18-6
- “Types of Target Code Formats” on page 18-10
- “Available Targets” on page 18-24
- “Selecting a Target ” on page 18-29
- “Template Makefiles and Make Options” on page 18-32
- “Custom Targets” on page 18-40

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.

In prior releases, 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` or `RealTimeMalloc`).

In Release 14, 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. The GRT target (and derived targets) generates a `RealTime` code format wrapper around the Embedded-C code. This wrapper provides a calling interface that is backward compatible with existing GRT-based custom targets. The wrapper calls are compatible with the main program module of the GRT target (`grt_main.c` or `grt_main.cpp`). This use of wrapper calls incurs some calling overhead; the pure Embedded-C calling interface generated by the ERT target is more highly optimized.

For a description of the calling interface generated by the ERT target, see “Developing Models for Code Generation” in the Real-Time Workshop Embedded Coder documentation.

Code format unification simplifies the conversion of GRT-based custom targets to ERT-based targets. See “Making GRT-Based Targets ERT-Compatible” on page 18-20 for a discussion of target conversion issues.

Targets and Code Formats

The Real-Time Workshop product provides five different *code formats*. Each code format specifies a framework for code generation suited for specific applications. The five code formats and corresponding application areas are

- **Real-time** — Rapid prototyping
- **Real-time malloc** — 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, real-time malloc, S-function, model referencing, and embedded C/C++ code formats.

Targets and Code Styles

The Real-Time Workshop 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 Real-Time Workshop 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 18-10.

Code Styles Listed by Target

Target	Code Style (Using C or C++ Unless Noted)
Real-Time Workshop 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 <i>production code target</i>).
Real-Time Workshop 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. Uses components of ERT, with a different calling interface.
Real-time malloc target	Rapid prototyping — Similar to the generic real-time (GRT) target except that this target allocates all model working memory dynamically rather than statically declaring it in advance.
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.

Code Styles Listed by Target (Continued)

Target	Code Style (Using C or C++ Unless Noted)
Tornado (VxWorks) real-time target ⁴	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.
xPC Target	Rapid prototyping — Runs model in real time on target PC running the xPC Target kernel.

Third-party vendors supply additional targets for the Real-Time Workshop 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.

4. Tornado® and VxWorks® are registered trademarks of Wind River® Systems, Inc.

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 any *model.h* file 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 all Real-Time Workshop 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 Real-Time Workshop license with those available under a Real-Time Workshop Embedded Coder license:

Comparison of Features Licensed with the Real-Time Workshop Product Versus the Real-Time Workshop Embedded Coder Product

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
rtModel data structure	<ul style="list-style-type: none"> Full rtModel structure generated GRT variable declaration: <code>rtModel_model model_M_;</code> 	<ul style="list-style-type: none"> rtModel is optimized for the model Optional suppression of error status field and data logging fields ERT variable declaration: <code>RT_MODEL_model model_M_;</code>
Custom storage classes (CSCs)	Code generation ignores CSCs; objects are assigned a CSC default to Auto storage class	Code generation with CSCs is supported
HTML code generation report	Basic HTML code generation report	Enhanced report with additional detail and hyperlinks to the model
Symbol formatting	Symbols (for signals, parameters and so on) are generated in accordance with hard-coded default	Detailed control over generated symbols.
User-defined maximum identifier length for generated symbols	Supported	Supported
Generation of terminate function	Always generated	Option to suppress terminate function
Combined output/update function	Separate output/update functions are generated	Option to generate combined output/update function
Optimized data initialization	Not available	Options to suppress generation of unnecessary initialization code for zero-valued memory, I/O ports, and so on

Comparison of Features Licensed with the Real-Time Workshop Product Versus the Real-Time Workshop Embedded Coder Product (Continued)

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
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 the Real-Time Workshop Embedded Coder documentation)
Target-optimized data types header file	Requires full <code>tmwtypes.h</code> header file	Generates optimized <code>rtwtypes.h</code> header file, including only the necessary 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
Simplified call interface	Non-ERT targets default to GRT interface	ERT and ERT-based targets generate simplified interface
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)
MAT-file logging	No option to suppress MAT-file logging data structures	Option to suppress MAT-file logging data structures
Reusable (multi-instance) code generation with static memory allocation	Not supported	Option to generate reusable code
Software constraint options	Support for floating point, complex, and nonfinite numbers is always enabled	Options to enable or disable support for floating-point, complex, and nonfinite numbers

Comparison of Features Licensed with the Real-Time Workshop Product Versus the Real-Time Workshop Embedded Coder Product (Continued)

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
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
ANSI ⁵ -C/C++ code generation	Supported	Supported
ISO ⁶ -C/C++ code generation	Supported	Supported
GNU ⁷ -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

-
- 5. ANSI® is a registered trademark of the American National Standards Institute, Inc.
 - 6. ISO® is a registered trademark of the International Organization for Standardization.
 - 7. GNU® is a registered trademark of the Free Software Foundation.

Types of Target Code Formats

In this section...

- “Real-Time Code Format” on page 18-14
- “Real-Time malloc Code Format” on page 18-15
- “S-Function Code Format” on page 18-17
- “Embedded Code Format” on page 18-18

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` or `RealTime malloc` 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, the Embedded-C code format is appropriate. Consider using the Real-Time Workshop 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, GRT-Malloc, ERT, xPC Target, Wind River Systems Tornado, Real-Time Windows Target, Texas Instruments™ DSP, Freescale™ MPC5xx, ...

The following table summarizes the various options available for each Real-Time Workshop code format/target, with the exceptions noted.

Features Supported by Real-Time Workshop Targets and Code Formats

Feature	GRT	Real-time malloc	ERT	ERT Shared Library	Wind River Systems VxWorks /Tornado	S-Func	RSIM	RT Win	xPC	Other Supported Targets ¹
Static memory allocation	X		X		X			X	X	X
Dynamic memory allocation		X			X	X	X		X	
Continuous time	X	X	X		X	X	X	X	X	

Features Supported by Real-Time Workshop Targets and Code Formats (Continued)

Feature	GRT	Real-time malloc	ERT	ERT Shared Library	Wind River Systems VxWorks /Tornado	S-Func	RSIM	RT Win	xPC	Other Supported Targets ¹
C/C++ MEX S-functions (noninlined)	X	X	X		X	X	X	X	X	
S-function (inlined)	X	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	X	
Rapid prototyping	X	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						
Executes in hard real time	X ⁴	X ⁴	X ⁴		X			X	X	X ⁵

Features Supported by Real-Time Workshop Targets and Code Formats (Continued)

Feature	GRT	Real-time malloc	ERT	ERT Shared Library	Wind River Systems VxWorks /Tornado	S-Func	RSIM	RT Win	xPC	Other Supported Targets ¹
Non real-time executable included	X	X	X				X			
Multiple instances of model		X ⁶	X ^{6, 7}			X ⁶			X ^{2, 6, 7}	X ^{2, 6, 7}
Supports variable-step solvers						X	X			
Supports SIL/PIL			X							X

¹The products that support other targets are: Embedded IDE Link, Target Support Package, Target Support Package, Embedded IDE Link, Target Support Package, and Target Support Package.

²Does not apply to GRT based targets. Applies only to an ERT based target.

³Except MPC5xx (algorithm export) targets

⁴The default GRT, GRT malloc, 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.

⁵Except MPC5xx (processor-in-the-loop) and MPC5xx (algorithm export) targets

⁶You 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.

⁷You must enable **Generate reusable code** in the Configuration Parameters **Real-Time Workshop – Interface** pane.

Real-Time Code Format

- “Introduction” on page 18-14
- “Unsupported Blocks” on page 18-14
- “System Target Files” on page 18-15
- “Template Makefiles” on page 18-15

Introduction

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 Chapter 32, “Integrating External Code Using S-Functions”, and the Target Language Compiler documentation.

The real-time code format declares memory statically, that is, at compile time.

Unsupported Blocks

The real-time format does not support the following built-in user-defined blocks:

- MATLAB Fcn 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++
 - grt_watc.tmf — Watcom C
- rsim
 - rsim_lcc.tmf — Lcc compiler
 - rsim_unix.tmf — UNIX host
 - rsim_vc.tmf — Visual C++®
 - rsim_watc.tmf — Watcom C
- tornado.tmf
- win_watc.tmf

Real-Time malloc Code Format

- “Introduction” on page 18-16
- “Unsupported Blocks” on page 18-16
- “System Target Files” on page 18-17

- “Template Makefiles” on page 18-17

Introduction

The real-time `malloc` code format (corresponding to the generic real-time `malloc` target) is very similar to the real-time code format. The differences are

- Real-time `malloc` declares memory dynamically.
For blocks provided by The MathWorks, `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.
- Real-time `malloc` allows you to deploy multiple instances of the same model with each instance maintaining its own unique data.
- Real-time `malloc` allows you to combine multiple models together in one executable. For example, to integrate two models into one larger executable, real-time `malloc` maintains a unique instance of each of the two models. If you do not use the real-time `malloc` format, the Real-Time Workshop code generator will not necessarily create uniquely named data structures for each model, potentially resulting in name clashes.

`grt_malloc_main.c` (or `.cpp`), the main routine for the generic real-time `malloc` (`grt_malloc`) target, supports one model by default. See “Combining Multiple Models” on page 3-114 for information on modifying `grt_malloc_main.c` (or `.cpp`) to support multiple models. `grt_malloc_main.c` and `grt_malloc_main.cpp` are located in the directory `matlabroot/rtw/c/grt_malloc`.

Unsupported Blocks

The real-time `malloc` format does not support the following built-in blocks, as shown:

- Functions & Tables
 - MATLAB Fcn 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_malloc.tlc` - Generic Real-Time Target with dynamic memory allocation
- `tornado.tlc` - Tornado (VxWorks) Real-Time Target

Template Makefiles

- `grt_malloc`
 - `grt_malloc_lcc.tmf` — Lcc compiler
 - `grt_malloc_unix.tmf` — The Open Group UNIX host
 - `grt_malloc_vc.tmf` — Microsoft Visual C++
 - `grt_malloc_watc.tmf` — Watcom 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 “Creating Component Object Libraries and Enhancing Simulation Performance” on page 3-92, for more information.

Embedded Code Format

- “Introduction” on page 18-18
- “Using the Real-Time Model Data Structure” on page 18-18
- “Making GRT-Based Targets ERT-Compatible” on page 18-20

Introduction

The Embedded-C code format corresponds to the Real-Time Workshop Embedded Coder target (ERT), and targets derived from ERT. This code format includes a number of memory-saving and performance optimizations. See the Real-Time Workshop Embedded Coder documentation for details.

Using 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 Real-Time Workshop code generator, and are documented in the Writing S-Functions documentation.

You need to use the set of macros `rtmGetxxx` and `rtmSetxxx` to access the real-time model data structure, which is specific to the Real-Time Workshop 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`, `grt_main.cpp`, `grt_malloc_main.c`, and `grt_malloc_main.cpp`.

Macros for Accessing the Real-Time Model Data Structure

rtm Macro Syntax	Description
rtmGetdX(rtM)	Get the derivatives of a block's continuous states
rtmGetOffsetTimePtr(RT_MDL rtM)	Return the pointer of vector that store all sample time offset 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 of NumSampleTime × NumSampleTime matrix
rtmGetRTWExtModeInfo(RT_MDL rtM)	Return an external mode information data structure of the model. This data structure is used internally for external mode.
rtmGetRTWLogInfo(RT_MDL)	Return a data structure used by Real-Time Workshop logging. Internal use.
rtmGetRTWRModelMethodsInfo(RT_MDL)	Return a data structure of Real-Time Workshop real-time model methods information. Internal use.
rtmGetRTWSolverInfo(RT_MDL)	Return data structure containing solver information of the model. Internal use.
rtmGetSampleHitPtr(RT_MDL)	Return a pointer of Sample Hit flag vector
rtmGetSampleTime(RT_MDL rtM, int TID)	Get a task's sample time
rtmGetSampleTimePtr(RT_MDL rtM)	Get pointer to a task's sample time
rtmGetSampleTimeTaskIDPtr(RT_MDL rtM)	Get pointer to a task's 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

Macros for Accessing the Real-Time Model Data Structure (Continued)

rtm Macro Syntax	Description
<code>rtmGetTFinal(RT_MDL)</code>	Get the simulation stop time
<code>rtmSetTFinal(RT_MDL, finalT)</code>	Set the simulation stop time
<code>rtmGetTimingData(RT_MDL)</code>	Return a data structure used by timing engine of the model. Internal use.
<code>rtmGetTPtr(RT_MDL)</code>	Return a pointer of the current time
<code>rtmGetTStart(RT_MDL)</code>	Get the simulation start time
<code>rtmIsContinuousTask(rtm)</code>	Determine whether a task is continuous
<code>rtmIsMajorTimeStep(rtm)</code>	Determine whether the simulation is in a major step
<code>rtmIsSampleHit(RT_MDL, tid)</code>	Determine whether the sample time is hit

For additional details on usage, see “SimStruct Functions — Alphabetical List” in the Writing S-Functions documentation.

Making GRT-Based Targets ERT-Compatible

If you have developed a GRT-based custom target, 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 a Real-Time Workshop Embedded Coder license, you can convert a GRT-based target as described in “Converting Your Target to Use `rtModel`” on page 18-21. This enables your custom target to support all current GRT features, including back end Embedded-C code generation.
- You can create an ERT-based target, but continue to use your customized version of the `grt_main.c` or `grt_main.cpp` module. To do this, you can configure the ERT target to generate a GRT-compatible calling interface, as described in “Generating GRT Wrapper Code from the ERT target” on

page 18-23. This lets your target support the full ERT feature set, without changing your GRT-based run-time interface. This approach requires that your installation has a Real-Time Workshop Embedded Coder license.

- If your installation includes a Real-Time Workshop Embedded Coder license, you can 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 “Creating and Using Custom Storage Classes” in the Real-Time Workshop Embedded Coder documentation for detailed information on CSCs.

For details on how GRT targets are made call-compatible with previous Real-Time Workshop product versions, see “Using the Real-Time Model Data Structure” on page 18-18.

Converting 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 all 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 all 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 `ssGetErrorHandler` to `rtmGetErrorHandler`.

Generating GRT Wrapper Code from the ERT target. The Real-Time Workshop Embedded Coder product supports the **GRT compatible call interface** option. When this option is selected, the Real-Time Workshop Embedded Coder product generates model function calls that are compatible with the main program module of the GRT target (`grt_main.c` or `grt_main.cpp`). These calls act as wrappers that interface to ERT (Embedded-C format) generated code.

This option provides a quick way to use ERT target features with a GRT-based custom target that has a main program module based on `grt_main.c` or `grt_main.cpp`.

See the “Preparing Models for Code Generation” in the Real-Time Workshop Embedded Coder documentation for detailed information on the **GRT compatible call interface** option.

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. All of these targets are built using the `make_rtw` make command.

Note You can select any target of interest 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 appropriate license on your system (Real-Time Workshop Embedded Coder license for ERT, Real-Time Windows Target license for RTWIN, and so on).

Each system target file invokes one or more template makefiles. The template makefile that is invoked activates a particular compiler (for example, Lcc, gcc, or Watcom compilers). This is specified for you by `MEXOPTS`, which is determined when you run `mex -setup` to select a compiler for `mex`. One exception is the Microsoft Visual C++ project target, which has separate System Target File Browser entries.

Targets Available from the System Target File Browser

Target/Code Format	System Target File	Template Makefile and Comments	Reference
Real-Time Workshop Embedded Coder (for PC or UNIX ⁸ platforms)	<code>ert.tlc</code> <code>ert_shrlib.tlc</code>	<code>ert_default_tmf</code> Use <code>mex -setup</code> to configure for Lcc, Watcom, vc, gcc, and other compilers	Real-Time Workshop Embedded Coder documentation

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

Targets Available from the System Target File Browser (Continued)

Target/Code Format	System Target File	Template Makefile and Comments	Reference
Real-Time Workshop Embedded Coder for Visual C++ ⁹ Project Makefile	ert.tlc	ert_msvc.tmf Creates a makefile which can be loaded into the Visual C++ IDE	Real-Time Workshop Embedded Coder documentation
Real-Time Workshop Embedded Coder for Tornado (VxWorks) ¹⁰	ert.tlc	ert_tornado.tmf	Real-Time Workshop Embedded Coder documentation
Real-Time Workshop Embedded Coder for AUTOSAR	autosar.tlc	ert_default_tmf	Real-Time Workshop Embedded Coder documentation
Generic Real-Time for PC or UNIX platforms	grt.tlc	grt_default_tmf Use mex -setup to configure for Lcc, Watcom, vc, gcc, and other compilers	Chapter 18, “Selecting and Configuring a Target”
Generic Real-Time for Visual C++ Project Makefile	grt.tlc	grt_msvc.tmf	Chapter 18, “Selecting and Configuring a Target”

9. Visual C++® is a registered trademark of Microsoft® Corporation.

10. Tornado® and VxWorks® are registered trademarks of Wind River® Systems, Inc.

Targets Available from the System Target File Browser (Continued)

Target/Code Format	System Target File	Template Makefile and Comments	Reference
Generic Real-Time (dynamic) for PC or UNIX platforms	grt_malloc.tlc	grt_malloc_default_tmf Use mex -setup to configure for Lcc, Watcom, vc, gcc, and other compilers Does not support SimStruct.	Chapter 18, “Selecting and Configuring a Target”
Generic Real-Time (dynamic) for Visual C++ Project Makefile	grt_malloc.tlc	grt_malloc_msvc.tmf Does not support SimStruct.	Chapter 18, “Selecting and Configuring a Target”
Rapid Simulation Target (default for PC or UNIX platforms)	rsim.tlc	rsim_default_tmf Use mex -setup to configure	Chapter 42, “Testing and Refining Concept Models With Standalone Rapid Simulations”
Rapid Simulation Target for LCC compiler	rsim.tlc	rsim_lcc.tmf	Chapter 42, “Testing and Refining Concept Models With Standalone Rapid Simulations”
Rapid Simulation Target for UNIX platforms	rsim.tlc	rsim_unix.tmf	Chapter 42, “Testing and Refining Concept Models With Standalone Rapid Simulations”
Rapid Simulation Target for Visual C++ compiler	rsim.tlc	rsim_vc.tmf	Chapter 42, “Testing and Refining Concept Models With Standalone Rapid Simulations”

Targets Available from the System Target File Browser (Continued)

Target/Code Format	System Target File	Template Makefile and Comments	Reference
Rapid Simulation Target for Watcom compiler	<code>rsim.tlc</code>	<code>rsim_watc.tmf</code>	Chapter 42, “Testing and Refining Concept Models With Standalone Rapid Simulations”
S-Function Target for PC or UNIX platforms	<code>rtwsfcn.tlc</code>	<code>rtwsfcn_default_tmf</code> Use <code>mex -setup</code> to configure	“Creating Component Object Libraries and Enhancing Simulation Performance” on page 3-92
S-Function Target for LCC	<code>rtwsfcn.tlc</code>	<code>rtwsfcn_lcc.tmf</code>	“Creating Component Object Libraries and Enhancing Simulation Performance” on page 3-92
S-Function Target for UNIX platforms	<code>rtwsfcn.tlc</code>	<code>rtwsfcn_unix.tmf</code>	“Creating Component Object Libraries and Enhancing Simulation Performance” on page 3-92
S-Function Target for Visual C++ compiler	<code>rtwsfcn.tlc</code>	<code>rtwsfcn_vc.tmf</code>	“Creating Component Object Libraries and Enhancing Simulation Performance” on page 3-92
S-Function Target for Watcom	<code>rtwsfcn.tlc</code>	<code>rtwsfcn_watc.tmf</code>	“Creating Component Object Libraries and Enhancing Simulation Performance” on page 3-92
Tornado (VxWorks) Real-Time Target	<code>tornado.tlc</code>	<code>tornado.tmf</code>	Chapter 34, “Interfacing With a Real-Time Operating System ”

Targets Available from the System Target File Browser (Continued)

Target/Code Format	System Target File	Template Makefile and Comments	Reference
ASAM-ASAP2 Data Definition Target	asap2.tlc	asap2_default_tmf	"Limitations on the Use of Absolute Time" on page 5-80
Real-Time Windows Target for Open Watcom	rtwin.tlc rtwinert.tlc	rtwin.tmf rtwinert.tmf	Real-Time Windows Target documentation
xPC Target for Visual C++ or Watcom C/C++ compilers	xpctarget.tlc xpctargetert.tlc	xpc_default_tmf xpc_ert_tmf xpc_vc.tmf xpc_watc.tmf	xPC Target documentation
Target Support Package	mpc555exp.tlc mpc555pil.tlc mpc555rt.tlc mpc555rt_grt.tlc	mpc555exp.tmf mpc555exp_diab.tmf mpc555pil.tmf mpc555pil_diab.tmf mpc555rt.tmf mpc555rt_grt.tmf	Target Support Package documentation
Target Support Package	c166.tlc c166_grt.tlc	c166.tmf c166_grt.tmf	Target Support Package documentation
Target Support Package	ccslink_grt.tlc ccslink_ert.tlc	N/A ¹²	Target Support Package documentation
Target Support Package	ccslink_grt.tlc (GRT) ccslink_ert.tlc	N/A ¹⁴	Target Support Package documentation

- 11. `ti_c2000_grt.tlc` (GRT) and `ti_c2000_ert.tlc` are provided for backward compatibility
- 12. `ti_c2000_grt.tmf` (GRT) and `ti_c2000_ert.tmf` are provided for backward compatibility
- 13. `ti_c6000.tlc` (GRT) and `ti_c6000_ert.tlc` are provided for backward compatibility
- 14. `ti_c6000.tmf` (GRT) and `ti_c6000_ert.tmf` are provided for backward compatibility

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 Real-Time Workshop software does not generate code and displays an error message. The Rapid Simulation (RSim) target supports a nonzero start time when the **Configuration Parameters > RSim Target > Solver selection** parameter is set to Use Simulink solver module. All other targets do not support a nonzero start time.

Selecting a Target

The first step to configuring a model for Real-Time Workshop code generation is to choose and configure a code generation target. When you select a target, other model configuration parameters change automatically to best serve requirements of the target. For example:

- Code interface parameters
- Build process parameters, such as the template make file
- Target hardware parameters, such as word size and byte ordering

Use the **Browse** button on the **Real-Time Workshop** pane to open the System Target File Browser (See “Selecting a Target” on page 18-29). The browser lets you select a preset target configuration consisting of a system target file, template makefile, and make command.

If you select a target configuration by using the System Target File Browser, your selection appears in the **System target file** field (`target.tlc`).

If you are using a target configuration that does not appear in the System Target File Browser, enter the name of your system target file in the **System target file** field. Click **Apply** or **OK** to configure for that target.

Chapter 18, “Selecting and Configuring a Target” describes the use of the browser and includes a complete list of available target configurations.

You can specify this configuration information for a specific type of target in one step by invoking the System Target File Browser, as explained in “Selecting a Target” on page 18-29. The browser lists a variety of ready-to-run configurations.

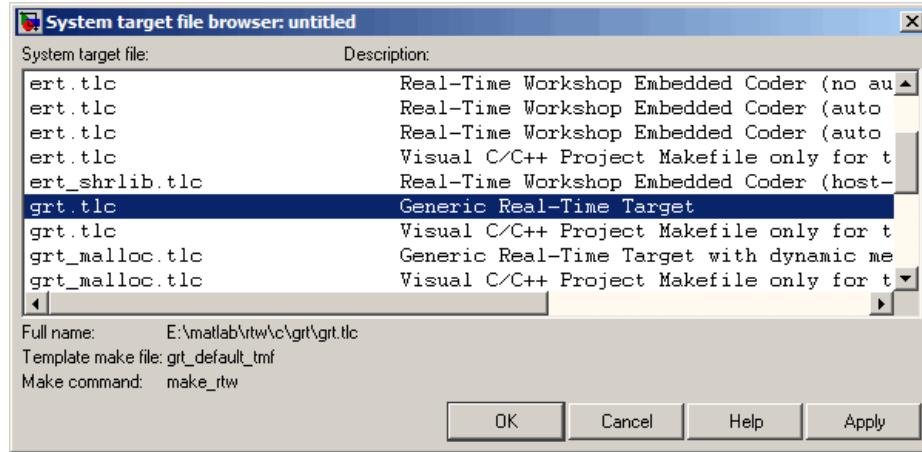
After selecting a system target, you can modify model configuration parameter settings, if necessary

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 “Setting Up Configuration Sets” in the Simulink documentation.

To select a target configuration using the System Target File Browser,

- 1 Click **Real-Time Workshop** on the Configuration Parameters dialog box. The **Real-Time Workshop** pane appears.
- 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 all currently available target configurations, including customizations. When you select a target configuration, the Real-Time Workshop software automatically chooses the appropriate system target file, template makefile, and make command.

“Selecting a Target” on page 18-29 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 Real-Time Workshop software automatically chooses the appropriate system target file, template makefile, and make command for the selected target, 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 general **Real-Time Workshop** pane.

Selecting 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 Real-Time Workshop system target file, template makefile, and 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 Real-Time Workshop 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 a Real-Time Workshop Embedded Coder program under the Lcc compiler.

- *Default* template makefiles make your model designs more portable, by choosing the correct compiler-specific makefile and compiler for your

installation. “Choosing and Configuring a Compiler” on page 25-2 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 appropriate TMF. 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 a Real-Time Workshop Embedded Coder program.

You can supply options to makefiles by using arguments to the **Make command** field in the general **Real-Time Workshop** pane of the Configuration Parameters dialog box. Append the arguments after `make_rtw` (or `make_xpc` 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 the Real-Time Workshop Embedded Coder documentation. This information is provided for those who want to customize template makefiles. This section describes compiler-specific template makefiles and common options you can use with each.

Note To control compiler optimizations for your Real-Time Workshop makefile build at the Simulink GUI level, use the **Compiler optimization level** option on the **Real-Time Workshop** 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 Real-Time Workshop 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®¹⁵ Std 1003.2-1992 (POSIX) standard.

- `ert_unix.tmf`
- `grt_malloc_unix.tmf`
- `grt_unix.tmf`
- `rsim_unix.tmf`
- `rtwsfcn_unix.tmf`

You can supply options by using arguments to the `make` command.

- `OPTS` — User-specific options, for example,

```
make_rtw OPTS="-DMYDEFINE=1"
```

15. IEEE® is a registered trademark of The Institute of Electrical and Electronics Engineers, Inc.

- **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 needed 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

The Real-Time Workshop product offers two sets of template makefiles designed for use with the Visual C++ compiler.

To build an executable within the Real-Time Workshop build process, use one of the **target_vc.tmf** template makefiles:

- **ert_vc.tmf**
- **grt_malloc_vc.tmf**
- **grt_vc.tmf**
- **rsim_vc.tmf**
- **rtwsfcn_vc.tmf**

You can supply options by using arguments to the **make** command.

- **OPT_OPTS** — Optimization option. Default is **-O2**. To enable debugging specify as **OPT_OPTS=-Zd**.
- **OPTS** — User-specific options.
- **CPP_OPTS** — C++ compiler options.
- **USER_SRCS** — Additional user sources, such as files needed 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 myfile.mak without building an executable, use one of the target_msvc.tmf template makefiles:

- ert_msvc.tmf
- grt_malloc_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 the following options by using arguments to the nmake command:

- OPTS — User-specific options, for example,

```
make_rtw OPTS="/D MYDEFINE=1"
```

- USER_SRCS — Additional user sources, such as files needed 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 Watcom C/C++ Compiler

The Real-Time Workshop product provides template makefiles to create an executable for the Microsoft Windows platform using Watcom C/C++. These template makefiles are designed to be used with wmake, which is bundled with Watcom C/C++.

Note The Watcom C compiler is no longer available from the manufacturer. However, the Real-Time Workshop product continues to ship with Watcom-related template makefiles.

- `ert_watc.tmf`
- `grt_malloc_watc.tmf`
- `grt_watc.tmf`
- `rsim_watc.tmf`
- `rtwsfcn_watc.tmf`

You can supply options by using arguments to the `make` command. Note that the location of the quotes is different from the other compilers and `make` utilities discussed in this chapter.

- `OPTS` — User-specific options, for example,

```
make_rtw "OPTS=-DMYDEFINE=1"
```

- `OPT_OPTS` — Optimization options. The default optimization option is `-oxat`. To turn off optimization and add debugging symbols, specify the `-d2` compiler switch in the `make` command, for example,

```
make_rtw "OPT_OPTS=-d2"
```

- `CPP_OPTS` — C++ compiler options.
- `USER_OBJS` — Additional user objects, such as files needed by S-functions.
- `USER_PATH` — The directory path to the source (`.c` or `.cpp`) files that are used to create any `.obj` files specified in `USER_OBJS`. Multiple paths must be separated with a semicolon. For example,

```
USER_PATH="path1;path2"
```

- `USER_INCLUDES` — Additional include paths, for example,

```
USER_INCLUDES="-Iinclude-path1 -Iinclude-path2"
```

These options are also documented in the comments at the head of the respective template makefiles.

Template Makefiles for the LCC Compiler

The Real-Time Workshop 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`
- `grt_malloc_lcc.tmf`
- `rsim_lcc.tmf`
- `rtwsfcn_lcc.tmf`

You can supply options by using arguments to the `make` command:

- `OPTS` — User-specific options, for example,

```
make_rtw OPTS="-DMYDEFINE=1"
```

- `OPT_OPTS` — Optimization options. Default is none. To enable debugging, specify `-g4` in the `make` command:

```
make_rtw OPT_OPTS="-g4"
```

- `CPP_OPTS` — C++ compiler options.
- `USER_SRCS` — Additional user sources, such as files needed by S-functions.
- `USER_INCLUDES` — Additional include paths, for example,

```
USER_INCLUDES="-Iwhere-ever -Iwhere-ever2"
```

For Lcc, have a / as file separator before the filename instead of a \, for example, `d:\work\proj1 myfile.c`.

These options are also documented in the comments at the head of the respective template makefiles.

Enabling the Real-Time Workshop Software to Build When Path Names Contain Spaces

The Real-Time Workshop software is able to handle path names that include spaces. Spaces might appear in the path from several sources:

- Your MATLAB installation directory
- The current MATLAB directory in which you initiate a build
- A compiler you are using for a Real-Time Workshop build

If your work environment includes one or more of the preceding scenarios, use the following support mechanisms, as necessary and appropriate:

- 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
```

This code replaces MATLAB_ROOT with ALT_MATLAB_ROOT when the values of the two tokens are not equal, indicating the path for your MATLAB installation directory includes spaces. Likewise, ALT_MATLAB_BIN replaces MATLAB_BIN.

Note the preceding code is specific to `nmake`. See the supplied Real-Time Workshop template make files for platform-specific examples.

- When using operating system commands, such as `system` or `dos`, enclose path 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. If you have in the past created system target files, note that the form of callbacks has changed between Versions 5 and 6 of the Real-Time Workshop product. See the Real-Time Workshop Embedded Coder documentation for details, including how to make your custom targets appear in the System Target File Browser and display appropriate controls in panes of the Configuration Parameters dialog box.

Selecting a Target Programming Language

Use the **Language** menu in the **Target selection** section of the **Real-Time Workshop** pane to select the target language for the code generated by the Real-Time Workshop code generator. You can select C or C++. The Real-Time Workshop software generates .c or .cpp files, depending on your selection, and places the files in your build directory.

Note If you select C++, you might need to configure the Real-Time Workshop software to use the appropriate compiler before you build a system. For details, see “Choosing and Configuring a Compiler” on page 25-2.

Specifying Code Appearance and Documentation

- “Configuring Code Comments” on page 20-2
- “Configuring Generated Identifiers” on page 20-3

Configuring Code Comments

Configure how the Real-Time Workshop code generator inserts comments into generated code, by modifying parameters on the Comments pane.

Note Comments can 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, and code generation template files.

To...	Select...
Include comments in generated code	Include comments. Selecting this parameter allows you to select one or more comment types to be placed in the code.
Include comments for blocks that were eliminated as the result of optimizations (such as parameter inlining)	Show eliminated blocks.
Automatically insert comments that describe a block's code before the code in the generated file	Simulink block / Stateflow object comments.
Include comments for parameter variable names and names of source blocks in the model parameter structure declaration in <i>model_prm.h</i>	Verbose comments for SimulinkGlobal storage class. 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. When you select the parameter, parameter comments are generated regardless of the number of parameters.

For descriptions of Comments pane parameters, see “Real-Time Workshop Pane: Comments” in the Real-Time Workshop reference documentation.

Configuring Generated Identifiers

In this section...

[“Reserved Keywords” on page 20-4](#)

[“Construction of Symbols” on page 20-7](#)

Configure how the Real-Time Workshop code generator uses symbols to name identifiers and objects by setting parameters on the **Symbols** pane.

Two options are available for GRT targets: **Maximum identifier length** and **Reserved names**. These are the only symbols options for GRT targets.

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. Consider increasing identifier length for models having a deep hierarchical structure, and when exercising some of the mnemonic identifier options described below.

Within a model containing Model blocks, no collisions of constituent model names can exist. When generating code from a model that uses model referencing, the **Maximum identifier length** must be large enough to accommodate the root model name and the name mangling string (if any). A code generation error occurs if **Maximum identifier length** is too small.

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.

The **Reserved names** field allows you to specify the set of keywords that the Real-Time Workshop code generation process should not use, facilitating code integration where functions and variables from external environments are unknown in the Simulink model. For a list of rules for specifying reserved names, see “Reserved names” in the Real-Time Workshop reference documentation.

If your model contains Embedded MATLAB Function or Stateflow blocks, the Real-Time Workshop code generation process can use the reserved names specified for those blocks if you select **Use the same reserved names as Simulation Target**.

If the Real-Time Workshop Embedded Coder product is installed on your system, the **Symbols** pane expands to include options for controlling identifier formats, mangle length, scalar inlined parameters, and Simulink data object naming rules. For details, see “Customizing Generated Identifiers” in the Real-Time Workshop Embedded Coder documentation.

For descriptions of Symbols pane parameters, see “Real-Time Workshop Pane: Symbols” in the Real-Time Workshop reference documentation.

Reserved Keywords

- “Real-Time Workshop C Keywords” on page 20-5
- “Real-Time Workshop C++ Keywords” on page 20-5
- “Real-Time Workshop Target Function Library Keywords” on page 20-6

Real-Time Workshop software reserves certain words for its own use as keywords of the generated code language. Real-Time Workshop keywords are reserved for use internal to Real-Time Workshop software and should not be used in Simulink models as identifiers or function names. If your model contains any reserved keywords, the Real-Time Workshop 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.

The following sections list Real-Time Workshop C Keywords and Real-Time Workshop C++ Keywords, and explain how to generate a list of Real-Time Workshop Target Function Library Keywords for a registered TFL.

If you are generating C++ code using the Real-Time Workshop software, your model must not contain both the Real-Time Workshop C Keywords and the Real-Time Workshop C++ Keywords.

Note You can register additional reserved identifiers in the Simulink environment. For more information, see “Reserved names” in the Real-Time Workshop reference documentation.

Real-Time Workshop C Keywords

abs	creal64_T	if	matrix	TRUE
asm	cuint8_T	int	NULL	typedef
auto	cuint16_T	int_T	pointer_T	uint_T
bool	cuint32_T	int8_T	real_T	uint8_T
boolean_T	default	int16_T	real32_T	uint16_T
break	do	int32_T	real64_T	uint32_T
byte_T	double	int64_T	register	uint64_T
case	else	localB	return	union
char	enum	localC	short	unsigned
char_T	extern	localDWork	signed	vector
cint8_T	false	localP	single	void
cint16_T	FALSE	localX	sizeof	volatile
cint32_T	float	localXdis	static	while
const	for	localXdot	struct	
continue	fortran	localZCE	switch	
creal_T	goto	localZCSV	time_T	
creal32_T	id_t	long	true	

Real-Time Workshop C++ Keywords

catch	friend	protected	try
class	inline	public	typeid
const_cast	mutable	reinterpret_cast	typename

<code>delete</code>	<code>namespace</code>	<code>static_cast</code>	<code>using</code>
<code>dynamic_cast</code>	<code>new</code>	<code>template</code>	<code>virtual</code>
<code>explicit</code>	<code>operator</code>	<code>this</code>	<code>wchar_t</code>
<code>export</code>	<code>private</code>	<code>throw</code>	

Real-Time Workshop Target Function Library Keywords

The list of target function library (TFL) reserved keywords for your development environment varies depending on which TFLs currently are registered. Beyond the default ANSI, ISO, and GNU TFLs provided with Real-Time Workshop software, additional TFLs might be registered and available for use if you have installed other products that provide TFLs (for example, a target product), or if you have used Real-Time Workshop Embedded Coder APIs to create and register custom TFLs.

To generate a list of reserved keywords for all TFLs currently registered in your environment, use the following MATLAB function:

```
tfl_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers()
```

This function returns an array of TFL keyword strings. Specifying the return argument is optional.

Note To list the TFLs currently registered in your environment, use the MATLAB command `RTW.viewTfl`.

To generate a list of reserved keywords for the TFL that you are using to generate code, call the function passing the name of the TFL as displayed in the **Target function library** menu on the **Interface** pane of the Configuration Parameters dialog box. For example,

```
tfl_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('GNU99 (GNU)')
```

Here is a partial example of the function output:

```
>> tfl_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('GNU99 (GNU)')
```

```

tfl_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.

Construction of Symbols

For GRT, GRT-malloc and RSim targets, the Real-Time Workshop code generator automatically constructs identifiers for variables and functions in the generated code. These symbols identify

- Signals and parameters that have Auto storage class
- Subsystem function names that are not user defined
- All Stateflow names

The components of a generated symbol 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 name mangling string is conditionally generated only when necessary to resolve potential conflicts with other generated symbols.

The length of generated symbols is limited by the **Maximum identifier length** parameter specified on the **Symbols** pane of the Configuration Parameters dialog box. When there is a potential name collision between two symbols, 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 in general. One way to do this is to avoid using default block names (for example, Gain1, Gain2...) when there are many blocks of the same type in the model. Also, whenever possible, make subsystems atomic and reusable.
- Where possible, increase the **Maximum identifier length** parameter to accommodate the length of the symbols you expect to generate.

Maximum identifier length can be longer for top model than referenced models. Model referencing can involve additional naming constraints. For information, see “Configuring Generated Identifiers” on page 20-3 and “Parameterizing Model References”.

Users of the Real-Time Workshop Embedded Coder product have additional flexibility over how symbols are constructed, by using a **Symbol format** field that controls the symbol formatting in much greater detail. See “Preparing Models for Code Generation” in the Real-Time Workshop Embedded Coder documentation for more information.

Customizing Targets

- “Overview of Embedded Target Development” on page 21-2
- “Example Custom Targets” on page 21-9
- “Target Development Mechanics” on page 21-11
- “Customizing System Target Files” on page 21-37
- “Customizing Template Makefiles” on page 21-86
- “Supporting Optional Features” on page 21-109
- “Interfacing to Development Tools” on page 21-130
- “Device Drivers and Target Preferences” on page 21-142

Overview of Embedded Target Development

In this section...

[“Introducing Custom Targets” on page 21-2](#)

[“Types of Targets” on page 21-2](#)

[“Recommended Features for Embedded Targets” on page 21-5](#)

Introducing Custom Targets

The targets bundled with the Real-Time Workshop 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 any 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 Real-Time Workshop 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 21-3
- “Rapid Prototyping Targets” on page 21-3
- “Turnkey Production Targets” on page 21-4
- “Verifying Targets With SIL and PIL Testing” on page 21-4

- “HIL Simulation Targets” on page 21-5

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 any 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 21-5, 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 Real-Time Workshop 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 Real-Time Workshop 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 “Example Custom Targets” on page 21-9.

For more information on SIL and PIL, see these topics in the Real-Time Workshop Embedded Coder documentation:

- “Verifying Generated Source Code With Software-in-the-Loop Simulation”
- “Verifying Compiled Object Code with Processor-in-the-Loop Simulation”

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 21-5
- “Integration with Target Development Environments” on page 21-6
- “Observing Execution of Target Code” on page 21-7
- “Deployment and Hardware Issues” on page 21-7

Basic Target Features

- You can base targets on the Real-Time Workshop generic real-time (GRT) target or the Embedded Real-Time (ERT) target that is included in the Real-Time Workshop 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, you ensure that your target has all the production code generation capabilities of the ERT target.

See “Customizing System Target Files” on page 21-37 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 Real-Time Workshop concepts of singletasking 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 the “Developing Models for Code Generation” chapter of the Real-Time Workshop Embedded Coder documentation.

- You should generate the target executable’s main program module, rather than using a static main module (such as the static `ert_main.c` module provided with the Real-Time Workshop Embedded Coder product). 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 the Real-Time Workshop Embedded Coder documentation for information on generated and static main program modules.

- Follow the guidelines in “Folder and File Naming Conventions” on page 21-11.

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. “Customizing Template Makefiles” on page 21-86 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 “Interfacing to Development Tools” on page 21-130 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 demo 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 Real-Time Workshop 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 Real-Time Workshop Embedded Coder product.

Another option is to support Simulink External Mode over a serial interface (RS-232). See the Chapter 37, “Communicating With Code Executing on a Target System Using Simulink External Mode” 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 “Integrating Device Drivers” on page 21-142.

- 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 21-23 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 “Interfacing to Development Tools” on page 21-130.

- 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. The ability to automatically switch between these files, depending on the RAM/FLASH option value, is also needed.
- 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.

Example 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 technical solution on this web page:

Solution 1-BHU00D — An example guide to developing an embedded target

This solution 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:

- Solution ID 1-9RXFT3 — An example Freescale S12X target using the Cosmic Compiler
- Solution ID 1-BHT815 — An example Freescale S12X target using the CodeWarrior® Compiler

Both these demo kits contain example models, code generation files, and instruction guides on generating and testing code for the processor. The Cosmic example demonstrates 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 demo 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 supported hardware Web page: <http://www.mathworks.com/products/rtwembedded/supportedio.html>

- 3** You can watch videos showing overviews of both the demo kits at the following links:

- http://www.mathworks.com/products/demos/rtwembedded/S12X_final_generic/
- http://www.mathworks.com/products/demos/rtwembedded/STR9_final_multi/

For another example target for the ARM9 (STR9) processor, see Solution ID 1-BBT4ID — An example ARM®9 (STR9) target using the GNU ARM Compiler and Hitex STR9-comStick.

If you have questions on specific targets, please email mytarget@mathworks.com.

The demo kits and this document describe Real-Time Workshop 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 Real-Time Workshop targets.

Target Development Mechanics

In this section...

- “Folder and File Naming Conventions” on page 21-11
- “Components of a Custom Target” on page 21-12
- “Key Directories Under the Target Root (`mytarget`)” on page 21-17
- “Key Files in the Target Folder (`mytarget/mytarget`)” on page 21-20
- “Additional Directories and Files for Externally Developed Targets” on page 21-28
- “Understanding and Using the Build Process” on page 21-29

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 all 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 subdirectories 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 21-12
- “Code Components” on page 21-13
- “Control Files” on page 21-15

Overview

The components of a custom target are files located in a hierarchy of directories. 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 21-11.

The components of a custom target include

- Code components: C source code that supervises and supports execution of generated model code.
- Control files:
 - A system target file (STF) to control the code generation process.
 - File(s) to control the building of an executable from the generated code. In a traditional make-based environment, a template makefile (TMF) generates a makefile for this purpose. Another approach is to generate project files in support of a modern integrated development environment (IDE) such as the Freescale Semiconductor CodeWarrior IDE.
 - Hook files: Optional TLC and .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.
- Target preferences files: These files define a *target preferences class* associated with your target. Your target preference class lets you create data objects that define and store properties associated with your target.

For example, you may want to store a user-defined path to a cross-compiler that is invoked by the build 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, demos, and target preferences available from the MATLAB **Start** menu.

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 Real-Time Workshop 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 on page 21-14 summarizes the execution support files.

Execution Support Files

You Provide...	The Real-Time Workshop 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 Real-Time Workshop 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 Real-Time Workshop `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 the Real-Time Workshop Embedded Coder main program, `ert_main.c`. The Real-Time Workshop Embedded Coder documentation details the structure of the Real-Time Workshop Embedded Coder execution support files and the execution of Real-Time Workshop Embedded Coder code, and provides guidelines for customizing `ert_main.c`.

- *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 Real-Time Workshop 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 Real-Time Workshop APIs. See the Real-Time Workshop documentation 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, Real-Time Workshop build process parses the **Make command** field of the **Real-Time Workshop** target configuration pane, expecting to find the name of a top-level MATLAB file command that controls the build process (as well as optional arguments to that command). The default top-level control file for the build process is `make_rtw.m`.

Normally, target developers do not need detailed knowledge of how `make_rtw` works. (The details that are necessary to target developers are described in

“Understanding and Using the Build Process” on page 21-29.) You should not customize `make_rtw.m`. The `make_rtw.m` file contains all 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 any 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 any 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 (`model1.rtw`). The Target Language Compiler reads `model1.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 “Customizing System Target Files” on page 21-37.

Template Makefile (TMF). A TMF provides information about your model and your development system. The Real-Time Workshop build process uses this information to create an appropriate makefile (`.mk` file) to build 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 Real-Time Workshop 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” on page 18-32 section of the Real-Time Workshop documentation.

The detailed structure of the TMF is described in “Customizing Template Makefiles” on page 21-86.

Hook Files. Real-Time Workshop 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.

To ensure that hook files are called correctly by the build process, they must follow well-defined naming and location requirements. “Folder and File Naming Conventions” on page 21-11 describes these requirements.

Key Directories Under the Target Root (`mytarget`)

- “Target Root Folder (`mytarget`)” on page 21-17
- “Target Folder (`mytarget/mytarget`)” on page 21-18
- “Target Block Folder (`mytarget/blocks`)” on page 21-18
- “Development Tools Folder (`mytarget/dev_tool1, mytarget/dev_tool2`)” on page 21-20
- “Target Preferences Folder (`mytarget/mytarget/@mytarget`)” on page 21-20
- “Target Source Code Folder (`mytarget/src`)” on page 21-20

Target Root Folder (`mytarget`)

This folder contains the key subdirectories for the target (see “Folder and File Naming Conventions” on page 21-11). You can also locate miscellaneous

files (such as a `readme` file) in the target root folder. The following sections describe required and optional subdirectories 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 the Target Folder (`mytarget/mytarget`)” on page 21-20 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 demo models and any 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 demo models into the standard **Demos** in the Help contents and **Start** button, 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 demo models. MATLAB software uses this information to place the demo in the appropriate place in the Help contents and Start button.

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 directories 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 any other files required to support this process.

For each development environment, you should provide a separate folder.

You should use the target preferences mechanism to store information about a user's choice of development environment or tool, paths to the installed development tools, and so on. Using target preferences data in this way lets your build process code select the appropriate development environment and invoke the appropriate compiler and other utilities. See the code excerpt in “`mytarget_default_tmf.m` Example Code” on page 21-98 for an example of how to use target preferences data for this purpose.

Target Preferences Folder (`mytarget/mytarget/@mytarget`)

If you create a target preferences class to store information about user preferences, you should store data class definition files and other files that support your target-specific preferences in `mytarget/mytarget/@mytarget`. The Simulink Data Class Designer creates the `@mytarget` folder automatically within the parent 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 any common source code and configuration code (such as boot and startup code).

Key Files in the Target Folder (`mytarget/mytarget`)

- “Introduction” on page 21-21

- “mytarget.tlc” on page 21-21
- “mytarget.tmf” on page 21-22
- “mytarget_default_tmf.m” on page 21-22
- “mytarget_settings.tlc” on page 21-22
- “mytarget_genfiles.tlc” on page 21-23
- “mytarget_main.c” on page 21-23
- “STF_make_rtw_hook.m” on page 21-23
- “STF_wrap_make_cmd_hook.m” on page 21-24
- “STF_rtw_info_hook.m (obsolete)” on page 21-26
- “info.xml” on page 21-27
- “mytarget_overview.html” on page 21-28

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 Real-Time Workshop Embedded Coder software.

“Customizing System Target Files” on page 21-37 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 “Customizing Template Makefiles” on page 21-86.

If your target development environment requires automation of a modern integrated development environment (IDE) rather than use of a traditional make utility, see “Interfacing to Development Tools” on page 21-130.

It is often necessary to create multiple template makefiles to support different development environments. See “Supporting Multiple Development Environments” on page 21-62 and “`mytarget_default_tmf.m` Example Code” on page 21-98 for information.

mytarget_default_tmf.m

This file is optional. You can implement a `mytarget_default_tmf.m` file to select the correct template makefile, based on user preferences. See “Setting Up a Template Makefile” on page 21-97.

mytarget_settings.tlc

This file is optional. Its purpose is to centralize global settings in the code generation environment. See “Using `mytarget_settings.tlc`” on page 21-57 for details.

mytarget_genfiles.tlc

This file is optional. `mytarget_genfiles.tlc` is useful as a central file from which to invoke any 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 21-60 for details.

mytarget_main.c

A main program module is required for your target. To provide a main module, you can either

- Modify the `ert_main.c` module provided by the Real-Time Workshop Embedded Coder software
- Generate `mytarget_main.c` or `.cpp` during the build process

The “Developing Models for Code Generation” chapter of the Real-Time Workshop Embedded Coder documentation contains a detailed description of the operation of `ert_main.c`. The chapter also contains guidelines for generating and modifying a main program module.

The “Optimizing Generated Code” chapter of the Real-Time Workshop Embedded Coder documentation describes how you can generate a customized 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.

The “Optimizing Generated Code” section of the Real-Time Workshop Embedded Coder documentation describes the operation of the `STF_make_rtw_hook.m` hook file in detail.

STF_wrap_make_cmd_hook.m

Use this file to override the default Real-Time Workshop behavior for selecting the appropriate compiler tool to be used in the build process.

By default, the Real-Time Workshop build process is based on makefiles. On PC hosts, the build process creates *model.bat*, an MS-DOS batch file. *model.bat* sets up the appropriate environment variables for the compiler, linker and other utilities, and invokes a `make` utility. The batch file, *model.bat*, obtains the required environment variable settings from the `MAKECMD` field in the template makefile. The standard Real-Time Workshop template makefiles support only standard compilers that build executables on the host system.

When developing an embedded target, you often need to override these defaults. Typically, you need to support one or more target-specific cross-development systems, rather than supporting compilers for the host system. The `STF_wrap_make_cmd_hook` mechanism provides a way to set up an environment specific to an embedded development tool.

Note that the naming convention for this file is *not* based on the target name. It is based on the concatenation of the system target filename, *STF*, with the string '`_wrap_make_cmd_hook`'.

Stub makefiles. Many modern cross-development systems, such as the Freescale Semiconductor CodeWarrior development environment, are based on project files rather than makefiles. If the interface to the embedded development system is not makefile based, one recommended approach is to create a stub makefile. When the build process invokes the stub makefile, no action takes place.

STF_wrap_make_cmd_hook Mechanism. A recommended approach to supporting non-host-based development systems is to provide a hook file that is called instead of the default host-based compiler selection.

To do this, create a `STF_wrap_make_cmd_hook.m` file. If this file exists, the build process calls it instead of the default compiler selection process. Make sure that:

- The file is on the MATLAB path.

- The filename is the name of your STF, prepended to the string '`_wrap_make_cmd_hook.m`'.
- The hook function implemented in the file follows the function prototype shown in the code example below.

A typical approach would be to write a `STF_wrap_make_cmd_hook.m` file that creates a MS-DOS batch file (`model.bat`). The batch file first sets up environment variables for the embedded target development system. Then, it invokes the embedded target's make utility on the generated makefile. The `STF_wrap_make_cmd_hook` function should return a system command that invokes `model.bat`.

This approach is shown in “Example `STF_wrap_make_cmd_hook` Function” on page 21-26.

Alternatively, any MS-DOS batch file can be created by `STF_wrap_make_cmd_hook`, and the function can return any command; it is not limited to `model.bat`. Like the `exit` case of the `STF_make_rtw_hook` mechanism, this provides the flexibility to invoke other utilities or applications.

Note that on a PC host, the Real-Time Workshop build process checks the standard output (`STDOUT`) for an appropriate build success string. By default, the string is

```
"### Created"
```

You can change this specifying a different `BUILD_SUCCESS` variable in the template makefile.

Example STF_wrap_make_cmd_hook Function.

```
function makeCmdOut = stfname_wrap_make_cmd_hook(args)
    makeCmd      = args.makeCmd;
    modelName    = args.modelName;
    verbose      = args.verbose;

    % args.compilerEnvVal not used
    cmdFile = ['.\',modelName, '.bat'];
    cmdFileFid = fopen(cmdFile,'wt');
    if ~verbose
        fprintf(cmdFileFid, '@echo off\n');
    end

    try
        prefs = RTW.TargetPrefs.load('mytarget.prefs');
        catch exception
            rethrow(exception);
        end

        fprintf(cmdFileFid, '@set TOOL_VAR1=%s\n', prefs.ImpPath);
        fprintf(cmdFileFid, '@set TOOL_VAR2=x86-win32\n');
        toolRoot = fullfile(prefs.ImpPath,'host','tool','4.4b');
        fprintf(cmdFileFid, '@set TOOL_VAR3=%s\n', toolRoot);
        path = getenv('Path');
        path1 = fullfile(prefs.ImpPath,'host','license;');
        if ~isempty(strfind(path,path1)) path1 = ''; end
        fprintf(cmdFileFid, '@set Path=%s%s%s\n', path1, path);
        fullMakeCmd = fullfile(prefs.ImpPath,'host','tool',...
            'bin', makeCmd);
        fprintf(cmdFileFid, '%s\n', fullMakeCmd);
        fclose(cmdFileFid);
        makeCmdOut = cmdFile;
```

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 all properties that were formerly specified in your `STF_rtw_info_hook` file.

For backward compatibility, existing `STF_rtw_info_hook` files continue to operate correctly. However, you should convert your target and models to use the **Hardware Implementation** pane. See the “Configuring the Hardware Implementation” on page 33-2 section of the Real-Time Workshop User’s Guide.

info.xml

This file provides information to MATLAB software that specifies where to display the target toolbox on the MATLAB **Start** button menu.

Example info.xml File. This example shows you how to set up access to a target’s demo page and target preferences GUI from the MATLAB **Start** button.

```
<productinfo>

<matlabrelease>13</matlabrelease>
<name>Embedded Target for MYTARGET</name>
<type>simulink</type>
<icon>$toolbox/simulink/simulinkicon.gif</icon>

<list>

<listitem>
<label>Demos</label>
<callback>demo simulink 'Embedded Target for MYTARGET'</callback>
<icon>$toolbox/matlab/icons/demoicon.gif</icon>
</listitem>

<listitem>
<label>MYTARGET Target Preferences</label>
```

```
<callback>mytargetTargetPrefs =
RTW.TargetPrefs.load('mytarget.prefs');
gui(mytargetTargetPrefs); </callback>
<icon>$toolbox/simulink/simulink/simulinkicon.gif</icon>
</listitem>

</list>
</productinfo>
```

mytarget_overview.html

By convention, this file serves as home page for the target demos.

The `<description>` field in `demos.xml` should point to `mytarget_overview.html` (see “`mytarget/blocks/demos.xml`” on page 21-19).

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 Demonstration Model</p>

<p>This demo 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 Directories and Files for Externally Developed Targets

- “Introduction” on page 21-29
- “`mytarget/mytarget/mytarget_setup.m`” on page 21-29
- “`mytarget/mytarget/doc`” on page 21-29

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 the necessary 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 all documentation related to your target in the folder `mytarget/mytarget/doc`.

Understanding and Using the Build Process

- “Introduction” on page 21-30
- “Build Process Phases and Information Passing” on page 21-30
- “Additional Information Passing Techniques” on page 21-33

Introduction

To develop an embedded target, you need a thorough understanding of the Real-Time Workshop build process. Your embedded target uses the build process and may require you to modify or customize the process. A general overview of the build process is given in the “How to Develop an Application Using Real-Time Workshop Software” chapter of the Real-Time Workshop Getting Started Guide.

This section supplements that overview with a detailed flowchart of the build process as customized by the Real-Time Workshop 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 21-33, 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. All information in *model.rtw* is available to target TLC files.
- The Real-Time Workshop 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 target preferences data. Target preferences provide persistent information about the target, such as the location of your development tools.
- 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 (`s1build.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, information may be needed from the various sources mentioned above. For example, during the TLC phase, it may be necessary to execute a MATLABfile to obtain information from the MATLAB environment. Also, a given phase may generate information that is needed in a subsequent phase.

See “Key Files in the Target Folder (mytarget/mytarget)” on page 21-20 for details on the available MATLAB file and TLC hooks for information passing, with code examples.

21 Customizing Targets

21-32

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 Real-Time Workshop 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 Real-Time Workshop options and their values available in the TLC phase.

See “System Target File Structure” on page 21-38 for further information.

makevariable Field in rtwoptions Structure. Similarly, Real-Time Workshop 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 21-38 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 correctly, 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. This section describes two techniques for supplying this information.

The simpler, more traditional approach is to 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.

This approach is not satisfactory in an environment where the MATLAB installation is on a network and multiple users share read-only TMFs. Another possible drawback to this approach is that the tool information is only available during the makefile processing phase of the build process.

A second approach is to use the target preferences feature together with the *STF_wrap_make_cmd_hook* mechanism (see “*STF_wrap_make_cmd_hook* Mechanism” on page 21-24). In this approach, compiler and other tool path information is stored as preferences data, which is obtained by the *STF_wrap_make_cmd_hook.m* file. This allows tool path information to be saved separately for each user.

Another advantage to the second approach is that target preferences data is available to all phases of the build process, including the TLC phase. This information may be required to support features such as RAM/ROM profiling.

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 “Application Data” section of the MATLAB Creating Graphical User Interfaces document.

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. Make sure that `tlc2appdata.m` is stored in a folder on the MATLAB path.
- 2** Create the TLC file as shown. Save it as `test.tlc`.
- 3** Enter the following command at the MATLAB prompt to execute the TLC file:

```
tlc test.tlc
```

- 4** Get the application data at the MATLAB prompt:

```
k = getappdata(0, 'z80')
```

The function returns the value 314159.

5 Enter the following command.

```
who
```

Note that application data is not stored in the MATLAB workspace. Also observe that the z80 data is not visible. Using application data in this way has the advantage that it does not clutter the MATLAB workspace. Also, it helps prevent you from accidentally deleting your data, since it is not stored directly in the your workspace.

A real-world use of application data might be to collect information from the *model.rtw* file and store it for use later in the build process.

Adding Block-Specific Information to the Makefile. The `rtwmakecfg` mechanism provides a method for inlined S-functions such as driver blocks to add information to the makefile. This mechanism is described in “Using the `rtwmakecfg.m` API to Customize Generated Makefiles” on page 21-101.

Customizing System Target Files

In this section...

- “Controlling Code Generation With the System Target File” on page 21-37
- “System Target File Naming and Location Conventions” on page 21-38
- “System Target File Structure” on page 21-38
- “Defining and Displaying Custom Target Options” on page 21-47
- “Tips and Techniques for Customizing Your STF” on page 21-56
- “Tutorial: Creating a Custom Target Configuration” on page 21-71

Controlling 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 affecting the build process) and for displaying them in the Configuration Parameters dialog box
- A mechanism for inheriting options from another target (such as the Embedded Real-Time (ERT) target)

This chapter 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.

The Real-Time Workshop Target Language Compiler document and the "Functions and Scripts" section of the MATLAB documentation describe the features and syntax of both the TLC and MATLAB languages.

While reading this chapter, you may want to refer to the STFs provided with the Real-Time Workshop product. Most of these files are stored in the target-specific directories 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 properly 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 21-11.

System Target File Structure

- “Overview” on page 21-38
- “Header Comments” on page 21-41
- “TLC Configuration Variables” on page 21-42
- “TLC Program Entry Point and Related %includes” on page 21-43
- “RTW_OPTIONS Section” on page 21-44
- “rtwgensections Structure” on page 21-45
- “Additional Code Generation Options” on page 21-47
- “Model Reference Considerations” on page 21-47

Overview

This section is a guide to the structure and contents of an STF. The following listing shows the general structure of an STF. Note that this is not a complete code listing of an STF. The listing consists of excerpts from each of the sections that make up an STF.

```
%%-----  
%% Header Comments Section
```

```

%%-----
%% SYSTLC: Example Real-Time Target
%%     TMF: my_target.tmf MAKE: make_rtw EXTMODE: ext_comm
%% Initial comments contain directives for STF Browser.
%% Documentation, date, copyright, and other info may follow.
    ...
%selectfile NULL_FILE
    ...
%%-----
%% TLC Configuration Variables Section
%%-----
%% Assign code format, language, target type.
%%
%assign CodeFormat = "Embedded-C"
%assign TargetType = "RT"
%assign Language = "C"
%%
%%-----
%% (OPTIONAL) Import Target Settings
%%
%include "mytarget_settings.tlc"
%%
%%-----
%% 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 %
%-----%

...
%%-----
%% rtwgenseettings Structure
%%-----
%% Define suffix string for naming build directory here.
rtwgenseettings.BuildDirSuffix = '_mytarget_rtw'
%% (OPTIONAL) target inheritance declaration
rtwgenseettings.DerivedFrom = 'ert.tlc';
%% (OPTIONAL) r14 callback compatibility declaration
rtwgenseettings.Version = '1';
%% (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 The 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 any other comments or TLC statements.

The presence of the comments enables the Real-Time Workshop software to detect STFs. When the System Target File Browser is opened, the Real-Time Workshop software scans the MATLAB path for TLC files that have correctly 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 **Real-Time Workshop** 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 **Real-Time Workshop** 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`.

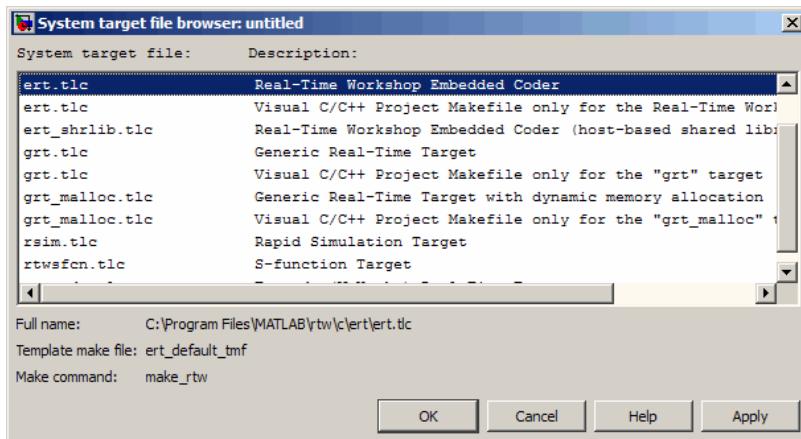
The following header comments are from `matlabroot/rtw/c/ert/ert.tlc`.

```
%% SYSTLC: Real-Time Workshop Embedded Coder (no auto configuration) \
%%      TMF: ert_default_tmf MAKE: make_rtw EXTMODE: ext_comm
.
.
.

%% SYSTLC: Visual C/C++ Project Makefile only for the Real-Time Workshop Embedded Coder\
%%      TMF: ert_msvc.tmf 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 last two lines specify a configuration that generates a Microsoft Visual C++ project makefile, using the template makefile `ert_msbc.tmf`. The figure below shows how these configurations appear in the System Target File Browser.



See “Tutorial: Creating a Custom Target Configuration” on page 21-71 for an example of customized header comments.

TLC Configuration Variables

This section of the STF assigns global TLC variables that affect the overall code generation process.

For an embedded target, in almost all 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. Make sure 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 the Chapter 18, “Selecting and Configuring a Target” chapter of the Real-Time Workshop documentation.

- **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 Real-Time Workshop documentation for more information).
- **TargetType:** The Real-Time Workshop 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 non-real-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 correctly assigned. Set these variables before including **codegenentry.tlc**.

If you need to implement target-specific code generation features, you should include the TLC files **mytarget_settings.tlc** and **mytarget_genfiles.tlc** in your STF. These files provide a mechanism for executing custom TLC code before and after invoking **codegenentry.tlc**. For information on these mechanisms, see

- “Using mytarget_settings.tlc” on page 21-57 for an example of custom TLC code for execution before the main code generation entry point.
- “Using mytarget_genfiles.tlc” on page 21-60 for an example of custom TLC code for execution after the main code generation entry point.
- “Understanding and Using the Build Process” on page 21-29 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 the *Real-Time Workshop Target Language Compiler* document for guidelines.

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 21-48.

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 21-45 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`. You must set:

```
rtwgensettings.Version = '1';
```

Add the statement above to the `Configure RTW code generation settings` section of the system target file.

If you are converting a target written prior to Release 14, do not use the `Version` field unless you have converted your callbacks, as described in “How to Convert Your `rtwOptions` Callbacks” on page 21-65.

- `rtwgensettings.DerivedFrom`: This string property defines the system target file from which options are to be inherited. See “Inheriting Target Options” on page 21-55.
- `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. When a model created prior to Release 14 is opened, the `SelectCallback` function is also triggered during model loading.

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, as described in “Using rtwoptions Callbacks Prior to Release 14” on page 21-64.

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 “Supporting Model Referencing” on page 21-110.

- `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, as described in “Using rtwoptions Callbacks Prior to Release 14” on page 21-64.

- `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, as described in “Using rtwoptions Callbacks Prior to Release 14” on page 21-64.

- `rtwgensettings.BuildDirSuffix`: Most targets define a string that identifies build directories 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 directories are named `model_mytarget_rtw`.

Additional Code Generation Options

“Configuring Generated Code with TLC” on page 27-19 in the Real-Time Workshop documentation describes additional TLC code generation variables. End users of any target can assign these variables by entering statements of the form

```
-aVariable=val
```

in the **TLC options** field of the **Real-Time Workshop** pane.

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 “Supporting Model Referencing” on page 21-110 for important information on STF and other modifications you may need to make to support the Real-Time Workshop model referencing features.

Defining and Displaying Custom Target Options

- “Using `rtwoptions` to Display Custom Target Options” on page 21-48
- “Example System Target File With Customized `rtwoptions`” on page 21-53
- “Inheriting Target Options” on page 21-55

Using `rtwoptions` to Display Custom Target Options

You control the options to display in the **Real-Time Workshop** pane of the Configuration Parameters dialog box by customizing the `rtwoptions` structure in your system target file.

The fields of the `rtwooptions` 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 **Real-Time Workshop** 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 **Real-Time Workshop** pane. See “NonUI Elements” on page 21-53 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 **Real-Time Workshop** pane. The header in each category is followed by one or more option definition elements.

The way in which target option groups are displayed depends on whether or not the STF has been converted for compatibility with Release 14 or later. In Release 14 or later compatible targets, each category of options corresponds to options listed under **Real-Time Workshop** in the Configuration Parameters dialog box. (See “Target Options Display Prior to Release 14” on page 21-69.)

The table `rtwoptions` Structure Fields Summary on page 21-51 summarizes the fields of the `rtwoptions` structure.

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 of three elements. The default field of the first (header) element is set to 4, indicating the number of elements that follow the header.

```
rtwoptions(1).prompt      = 'Real-Time Workshop S-Function Code Generation Options';
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).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 Real-Time Workshop 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 **S-function target options** under **Real-Time Workshop** 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 21-53.

`rtwoptions` Structure Fields Summary on page 21-51 lists the fields of the `rtwoptions` structure.

rtwoptions Structure Fields Summary

Field Name	Description
<code>callback</code>	<p>For examples of callback usage, see “Example System Target File With Customized <code>rtwoptions</code>” on page 21-53.</p> <p>If you need to convert callbacks for Release 14 or later compatibility, see “Upgrading Custom Targets to Release 14 or Later” on page 21-63.</p>
<code>closecallback</code> (obsolete)	<p>If your target uses <code>closecallback</code>, convert to <code>rtwgensettings.PostApplyCallback</code> instead (see “<code>rtwgensettings</code> Structure” on page 21-45).</p> <p>See “Upgrading Custom Targets to Release 14 or Later” on page 21-63 for information on converting callbacks for Release 14 or later compatibility.</p> <p><code>closecallback</code> is ignored in Release 14 or later. Prior to Release 14, <code>closecallback</code> specified a MATLAB code function to be executed when the target options dialog box closes.</p>

rtwoptions Structure Fields Summary (Continued)

Field Name	Description
	For examples of callback usage, see “Example System Target File With Customized rtwoptions” on page 21-53.
default	Default value of the option (empty if the type is Pushbutton).
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 21-87.
modelReferenceParameterCheck	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 21-116.
NonUI	Element that is not displayed, but is used to invoke a close or open callback. See “NonUI Elements” on page 21-53.
opencallback (obsolete)	If your target uses opencallback, we strongly recommend that you use rtwgenselections.SelectCallback instead (see “rtwgenselections Structure” on page 21-45). If you must maintain use of opencallback, see “Upgrading Custom Targets to Release 14 or Later” on page 21-63 for information on converting callbacks for Release 14 or later compatibility. For examples of callback usage, see also “Example System Target File With Customized rtwoptions” on page 21-53. Prior to Release 14, opencallback specified code to be executed when the selected the target from the System Target File Browser, or during model loading. The purpose of opencallback is to synchronize the displayed value of the option with its previous setting.

rtwoptions Structure Fields Summary (Continued)

Field Name	Description
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 any 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 21-53 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 **Real-Time Workshop** 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 are

- `usertarget.tlc`: The example system target file. This file defines several menus, check boxes, an edit field, and a nonUI item. The file demonstrates the use of callbacks, open callbacks, and closed callbacks.
- `usertargetcallback.m`: A MATLAB file callback invoked by a menu.
- `usertargetclosecallback.m`: MATLAB file callback invoked by an edit field.

Refer to the example files while reading this section. The example system target file, `usertarget.tlc`: demonstrates the use of callbacks associated with the following UI elements:

- The **Execution Mode** menu executes an open callback that is coded inline within the STF. This callback displays a message and sets a model property with a `set_param()`.
- The **Real-Time Interrupt Source** 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** executes a close callback defined in an external file, `usertargetclosecallback.m`. The TLC variable associated with the edit field is passed in to the callback.
- The **External Mode** check box executes an open callback that is coded inline within the STF.
- The **NonUi** item defined in `rtwoptions(8)` executes open and close callbacks that are coded inline within the STF. Each callback simply prints a status message.

We suggest that you study 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 any model of your choice.

- 3** Open the Configuration Parameters dialog box or Model Explorer and select the **Real-Time Workshop** pane.
- 4** Click **Browse**. The System Target File Browser opens. Select `usertarget.tlc` Real-Time Workshop Options Example Target. Then click **OK**.
- 5** Observe that the **Real-Time Workshop** 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 Real-Time Workshop 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 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.

We recommend that you handle these options in mytarget_settings.tlc. See the example in “Using mytarget_settings.tlc” on page 21-57.

Even though your target may not support all inherited ERT options, it is required that the ERT options are retained in the **Real-Time Workshop** pane of the GUI. Do not simply remove unsupported options from the **rtwoptions** structure in the STF. Options must be in the GUI to be scanned by the Real-Time Workshop 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 safe to remove this option from the GUI and simply assign the TLC variable **CombineOutputUpdateFcns** to on. However, if the option is not included in the GUI, the Real-Time Workshop 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 21-56
- “Required and Recommended %includes” on page 21-57
- “Handling Aliases for Target Option Values” on page 21-60
- “Supporting Multiple Development Environments” on page 21-62
- “Upgrading Custom Targets to Release 14 or Later” on page 21-63

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 with single or multiple STFs

Required and Recommended %includes

If you need to implement target-specific code generation features, we recommend that your STF include the TLC files `mytarget_settings.tlc` and `mytarget_genfiles.tlc`.

`mytarget_settings.tlc` provides a mechanism for executing custom TLC code before the main code generation entry point. See “Using `mytarget_settings.tlc`” on page 21-57.

Once your STF has set up any 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 21-60.

Using `mytarget_settings.tlc`. This file is optional. Its purpose is to centralize global settings in the code generation environment. Use `mytarget_settings.tlc` to

- Define required TLC paths with `%addincludepath` directives. You may need to do this if you create target-specific TLC function libraries.
- Create records that store target-specific path information and preference settings in the `CompiledModel` general record. This provides a clean mechanism for passing this information into the TLC code generation environment.
- Check user settings for code generation options. If incorrect or unsupported option settings are found, issue the appropriate error or warning and abort the build process if necessary.

mytarget_settings.tlc Example Code

In the TLC code example below, the structure `Settings` is added to the `CompiledModel` record. The `Settings` structure is loaded from the stored target preferences. The `Settings` structure stores target preferences data fields `Implementation` and `ImpPath`.

After `Settings` is added to the `CompiledModel` record, the example code handles inherited options. In this example, the target is assumed to have inherited options from the ERT target. The code examines the settings of inherited ERT code generation options. If the user has selected unsupported options, warning or error messages are displayed. In some cases, selecting an unsupported option causes the build process to terminate.

Conditional code at the end of the function allows display of the `Implementation` and `ImpPath` fields in the MATLAB Command Window.

```
%selectfile NULL_FILE

%% Read user preferences for the target and add to CompiledModel
%assign prefs = FEVAL("RTW.TargetPrefs.load","mytarget.prefs","structure")
%addtorecord CompiledModel Settings prefs

%% Check for unsupported Embedded Coder options and error/warn appropriately
%if SuppressErrorStatus == 0
    %assign SuppressErrorStatus = 1
    %assign msg = "Suppressing Error Status as it is not used by this target."
    %warning %<msg>
%endif
%if GenerateSampleERTMain == 1
    %assign msg = "Generating an example main is not supported as the proper main
function is inherently generated. Unselect the \"Generate an example main program\""
checkbox under ERT code generation options."
    %exit %<msg>
%endif

%if GenerateErtSFunction == 1
    %assign msg = "Generating a Simulink S-Function is not supported. Unselect the
\"Create SIL block\" checkbox under ERT code generation options."
    %exit %<msg>
```

```

%endif

%if ExtMode == 1
    %assign msg = "External Mode is not currently supported. Unselect the \"External
mode\" checkbox under ERT code generation options."
    %exit %<msg>
%endif

%if MatFileLogging == 1
    %assign msg = "MAT-file logging is not currently supported. Unselect the
\"MAT-file logging\" checkbox under ERT code generation options."
    %exit %<msg>
%endif

%if MultiInstanceERTCode == 1
    %assign msg = "Generate reuseable code is not currently supported. Unselect the
\"Generate reuseable code\" checkbox under ERT code generation options."
    %exit %<msg>
%endif

%if GenFloatMathFcnCalls == "ISO_C"
    %assign msg = "Target function libraries other than ANSI-C are not
currently supported. Select ANSI-C for the \"Target function library\" option
under ERT code generation options."
    %exit %<msg>
%endif

%% To display added TLC settings for debugging purposes, set EchoConfigSettings to
1.
%assign EchoConfigSettings = 0
%if EchoConfigSettings
    %selectfile STDOUT
    #####IMPLEMENTATION#####
    IMPLEMENTATION is:
    %<CompiledModel.Settings.Implementation>

    IMPLEMENTATION path is:
    %<CompiledModel.Settings.ImpPath>

```

```
#####
%selectfile NULL_FILE
%endif
```

Using mytarget_genfiles.tlc. `mytarget_genfiles.tlc` (optional) is useful as a central file from which to invoke any 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 “Customizing the Target Build Process with the STF_make_rtw Hook File” on page 27-22.

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 target function library (`GenFloatMathFcnCalls`).

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 options “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 options “Device vendor” and “Device type” in the Simulink reference documentation.

RTW.isTfEq. To test if two target function library (TFL) strings represent the same TFL, invoke the following function:

```
result = RTW.isTflEq(name1, name2)
```

where *name1* and *name2* are strings containing TFL values or aliases.

The `RTW.isTflEq` function returns true if *name1* and *name2* are strings representing the same TFL. For example, the following call returns true:

```
RTW.isTflEq('ANSI_C', 'C89/C90 (ANSI)')
```

For a description of the TFL option `GenFloatMathFcnCalls`, see the command-line information for the **Interface** pane option “Target function library” in the Real-Time Workshop reference documentation.

RTW.resolveTflName. To return the TFL value for a target function 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 TFL value or alias.

The `RTW.resolveTflName` function returns the TFL value of the referenced TFL. For example, the following calls both return '`C89/C90 (ANSI)`':

```
RTW.resolveTflName('ANSI_C')
RTW.resolveTflName('C89/C90 (ANSI)')
```

For a description of the TFL option `GenFloatMathFcnCalls`, see the command-line information for the **Interface** pane option “Target function library” in the Real-Time Workshop reference documentation.

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 an appropriate template makefile for the development environment. This amounts to providing two separate targets.

Another approach is to use a single STF that specifies multiple configurations in its comment header. The code within the STF then checks the target preferences to determine which template makefile to invoke. See “mytarget_default_tmf.m Example Code” on page 21-98 for an example of how to check target preferences for this information.

One drawback of using a single STF in this way is that the rtwoptions need conditional sections if the target options are not the same for all of the configurations the STF supports. The following example (from a hypothetical example target) defines an rtwoptions menu element differently, depending on the whether or not the MATLAB software is running on a PC (Microsoft Windows platform). This is determined by calling the MATLAB function `ispc`. On the PC, the menu displays a choice of USB or serial ports to be used in communicating with a target device. Otherwise, the menu displays a choice of UNIX¹⁶ logical devices.

```
if ispc
    rtwoptions(rtwoption_index).default      = 'USB';
    rtwoptions(rtwoption_index).popupstrings =
    'USB|COM1|COM2|COM3|COM4';
else
    rtwoptions(rtwoption_index).default      = '/dev/ttyS0';
    rtwoptions(rtwoption_index).popupstrings =
    '/dev/ttyS0|/dev/ttyS1|/dev/ttyS2|/dev/ttyS3';
end
```

Upgrading Custom Targets to Release 14 or Later

In Release 14 or later, you view the Simulink model options defined in the active configuration set using the Configuration Parameters dialog box or Model Explorer. These views, which replaced the Simulation Parameters dialog box used in releases before Release 14, feature extensive changes in the appearance and layout of code generation options and other target-specific options for Real-Time Workshop targets. For customers upgrading from a release before Release 14, this section describes the following compatibility issues related to the definition and display of target-specific options for custom targets:

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

- **rtwoptions callbacks:** If the `rtwoptions` array in your custom system target file contains callbacks, convert your callbacks to use the callback compatibility API provided in Release 14 or later.
 - **Target options inheritance:** If your custom target is derived from another target and inherits options, change your system target file to use the inheritance mechanism provided in Release 14 or later.
 - **Display of target options:** Your target options will display differently in Release 14 or later, and you may want to reorganize them.
-
- “Using `rtwoptions` Callbacks Prior to Release 14” on page 21-64
 - “How to Convert Your `rtwOptions` Callbacks” on page 21-65
 - “Example Callback Code” on page 21-66
 - “Operation of Targets with Unconverted Callbacks” on page 21-67
 - “Target Options Display Prior to Release 14” on page 21-69

Using `rtwoptions` Callbacks Prior to Release 14. In releases before Release 14, the `callback`, `opencallback`, and `closecallback` fields of the `rtwoptions` structure array (see “Using `rtwoptions` to Display Custom Target Options” on page 21-48) specify optional functions that are called when the value of an option changes or when the Simulation Parameters dialog box opens or closes. If your custom system target file does *not* specify any such callbacks, your target operates transparently in the Configuration Parameters dialog box. However, your target options are displayed differently, as described in “Target Options Display Prior to Release 14” on page 21-69.

If your custom target does specify callbacks, compatibility issues arise, because many callbacks depend upon features of the old-style (pre-Release 14) Simulation Parameters dialog box. For example, a change in the state of one GUI element (such as a check box) may invoke a callback that attempts to get a handle to another GUI element in order to enable or disable it.

In Release 14 or later, the Real-Time Workshop software supports a callback compatibility API that lets your existing `rtwoptions` callbacks operate under the Configuration Parameters dialog box. This is described in the next section, “How to Convert Your `rtwOptions` Callbacks” on page 21-65. We strongly recommend that you convert your callbacks for Release 14 or later compatibility. If you do not want to do so, see “Operation of Targets with

“Unconverted Callbacks” on page 21-67 to understand how your custom target runs in the Release 14 or later environment.

How to Convert Your `rtwOptions` Callbacks. The callback conversion API provides variables and accessor functions that allow your callbacks to access graphical elements associated with target options. Also, a version compatibility property, `rtwgensettings.Version`, has been added to the `rtwgensettings` structure in the system target file. This property identifies targets that have been converted to use Release 14 or later compatible callbacks.

The callback API variables are

- `model`: Handle of the current Simulink model. `model` can be used as an argument to `get_param` and `set_param` calls. If you use such calls, you do not need to change them.
- `hSrc`: This variable is restricted to use in the callback API functions described below. `hSrc` provides a handle to private data used by the callback API functions. Do not set this variable or use it for any other purpose.
- `hDlg`: This variable is restricted to use in the callback API functions described below. `hDlg` provides a handle to private data used by the callback API functions. Do not set this variable or use it for any other purpose.

The callback API provides accessor functions that let you read and set target option values, and enable or disable options. In the function descriptions below, the `tlc_var_name` argument is the name of the `tlcvariable` defined for the option in the `rtwoptions` struct. The callback API accessor functions are

- `slConfigUIGetVal(hDlg, hSrc, 'tlc_var_name')`: Returns the current value of the option specified by the '`tlc_var_name`' argument. The data type of the return value depends on the data type of the option.
- `slConfigUISetVal(hDlg, hSrc, 'tlc_var_name', value)`: Sets the option specified by the '`tlc_var_name`' argument to the value passed in the `value` argument.

- `slConfigUISetEnabled(hDlg, hSrc, 'tlc_var_name', flag)`: Enables or disables the option specified by the '`'tlc_var_name'`' argument. The value passed in `flag` should be either 1 (to enable the option) or 0 (to disable the option).

To convert your `rtwOptions` callbacks,

- 1 Identify all references to the old Simulation Parameters dialog box handle (such as `dialogFig` or objects accessed through `dialogFig`) in your callbacks.
- 2 Replace such references with equivalent calls to the callback API functions. Your code should use only the API calls and variables described above to reference options. See the files described in “Example Callback Code” on page 21-66.
- 3 If your target inherits options from an existing target, you should also convert your target to use the new inheritance mechanism. To learn how to do this, see “Inheriting Target Options” on page 21-55.
- 4 Declare that your system target file is compliant with the callback API by adding the following statement in the `Configure RTW code generation settings` section of the system target file.

```
rtwgensettings.Version = '1';
```

`rtwoptions` callbacks are executed only if `rtwgensettings.Version` is set as shown.

Note If your target defines `opencallback` functions, open callbacks are called during model loading, as well as when you select the target from the System Target File Browser.

Example Callback Code. An example system target file and callback handlers are available in the folder `matlabroot/toolbox/rtw/rtwdemos/rtwoptions_demo`. The example files illustrate how to use Release 14 or later compatible callbacks with different types of GUI elements. The files are

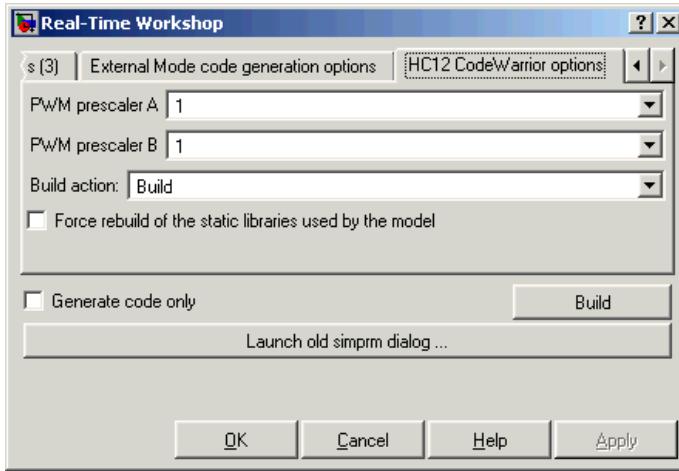
- `usertarget.tlc`: The example system target file. This file defines several menus, check boxes, an edit field, and a nonUI item. The file demonstrates the use of callbacks, open callbacks, and close callbacks.
- `usertargetcallback.m`: A callback invoked by a popup.
- `usertargetclosecallback.m`: A callback invoked by an edit field.

Operation of Targets with Unconverted Callbacks. Callback conversion is recommended, the **Launch old simprm dialog** button will be removed in a future release. If you do not want to convert your callbacks, your target operates as follows:

- When the target is selected with the System Target File Browser, the target options are displayed in the Configuration Parameters dialog box, as described in “Target Options Display Prior to Release 14” on page 21-69. However, any callbacks specified in the `rtwoptions` array are ignored.
- An additional button labeled **Launch old simprm dialog** is displayed at the bottom of all target-specific pages of the Configuration Parameters dialog box. When the user clicks this button, the old Simulation Parameters dialog box opens. As the user interacts with the dialog box, existing callbacks are executed.

Note The **Launch old simprm dialog** button will be removed in a future release. It is recommended that you perform the callback conversion.

The figure below shows the Model Explorer view.



Note If your custom target uses unconverted callbacks, you should inform end users of your target that they should open and use the old Simulation Parameters dialog box when setting target options. If they do not do so, options are not set correctly. Keep in mind, it is now recommended that you perform the callback conversion. The **Launch old simprm dialog** button will be removed in a future release.

In releases before Release 14, many custom targets used the technique of merging `rtwoptions` structures in order to derive or inherit options from an existing target. For example, the following code, from a Release 13 target, creates an `rtwoptions` structure and inherits the `rtwoptions` of the ERT target merging them into the structure.

```
/%
BEGIN_RTW_OPTIONS
rtwoption_index = 0;

rtwoption_index = rtwoption_index + 1;
rtwoptions(rtwoption_index).prompt      = 'mytargets Options';
rtwoptions(rtwoption_index).type        = 'Category';
rtwoptions(rtwoption_index).enable      = 'on';
rtwoptions(rtwoption_index).default     = 5;    % number of items under mytargets
rtwoptions(rtwoption_index).popupstrings = '';
rtwoptions(rtwoption_index).tlcvariable  = '';
```

```

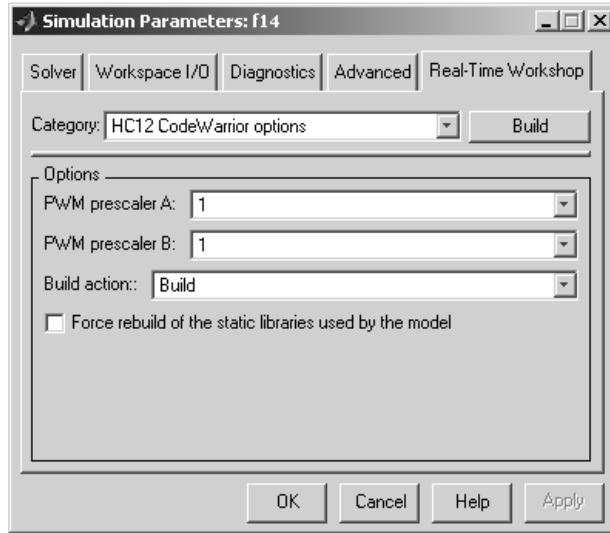
rtwoptions(rtwoption_index).tooltip      = '';
rtwoptions(rtwoption_index).callback     = '';
rtwoptions(rtwoption_index).makevariable = '';
%other rtwoptions elements not shown here
...
% Inherit ERT options
file      = fullfile(matlabroot, 'rtw', 'c', 'ert', 'ert.tlc');
propsObj = tlc.rtwoptions(file);
props    = propsObj.getOptions;
rtwoptions = propsObj.combineCategories(props, rtwoptions);

```

Releases 14 and later support a new, simplified inheritance mechanism. Remove old inheritance code (such as the four line after the `%Inherit ERT options` comment in the example above).

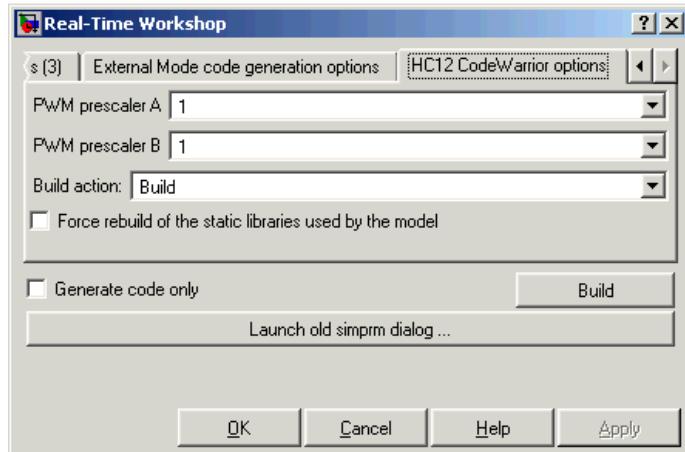
Target Options Display Prior to Release 14. In the Simulation Parameters dialog box present in releases before Release 14, target options are organized into functional groups, displayed under control of the **Category** menu in the **Real-Time Workshop** pane. The items in the **Category** menu correspond to the elements of the `rtwoptions` structure array. Each group of `rtwoptions` elements is delimited by a header element of type **Category**.

The following figure shows a typical group of target options as displayed in the old-style Simulation Parameters dialog box.

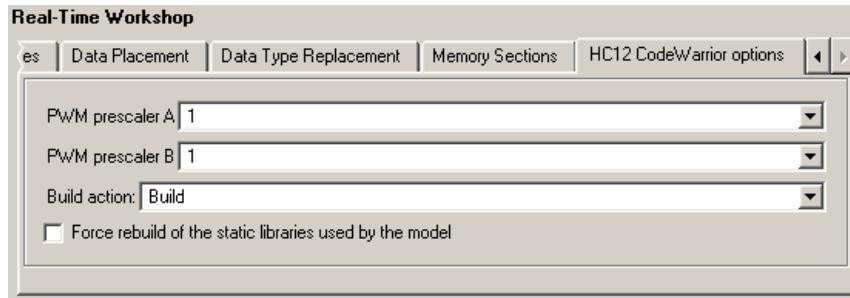


The new Configuration Parameters dialog box preserves the organization of your custom target's `rtwoptions` structure array. However, the **Category** menu has been replaced by a tabbed selection mechanism. In the Model Explorer view, each category of options corresponds to a tab. In the Configuration Parameters dialog box view, each category of options corresponds to an element of the list on the left pane. The spacing and layout of options within each group of options is controlled by the Real-Time Workshop software.

The figure below shows the same target options, as organized and displayed in the Model Explorer view. This figure shows how the target options appear before any Release 14 or later compatibility conversions are made.



After converting the above target to use Release 14 or later compatible callbacks and inheritance options, the target's inherited options are displayed in a more compact form (under categories such as **Interface**, **Templates**, and so on) and the **Launch old simprm dialog** button is removed, as shown in this figure.



The Real-Time Workshop product provides the organization of options described above as a default layout. This lets you continue to use your custom targets with minimal change. This default differs considerably from many of the targets developed internally at The MathWorks (such as the ERT and GRT targets).

Tutorial: Creating a Custom Target Configuration

- “Introduction” on page 21-71
- “my_ert_target Overview” on page 21-72
- “Creating Target Directories” on page 21-74
- “Create ERT-Based STF” on page 21-75
- “Create ERT-Based TMF” on page 21-82
- “Create Test Model and S-Function” on page 21-82
- “Verify Target Operation” on page 21-84

Introduction

This tutorial can supplement the example target guides described in “Example Custom Targets” on page 21-9. 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:

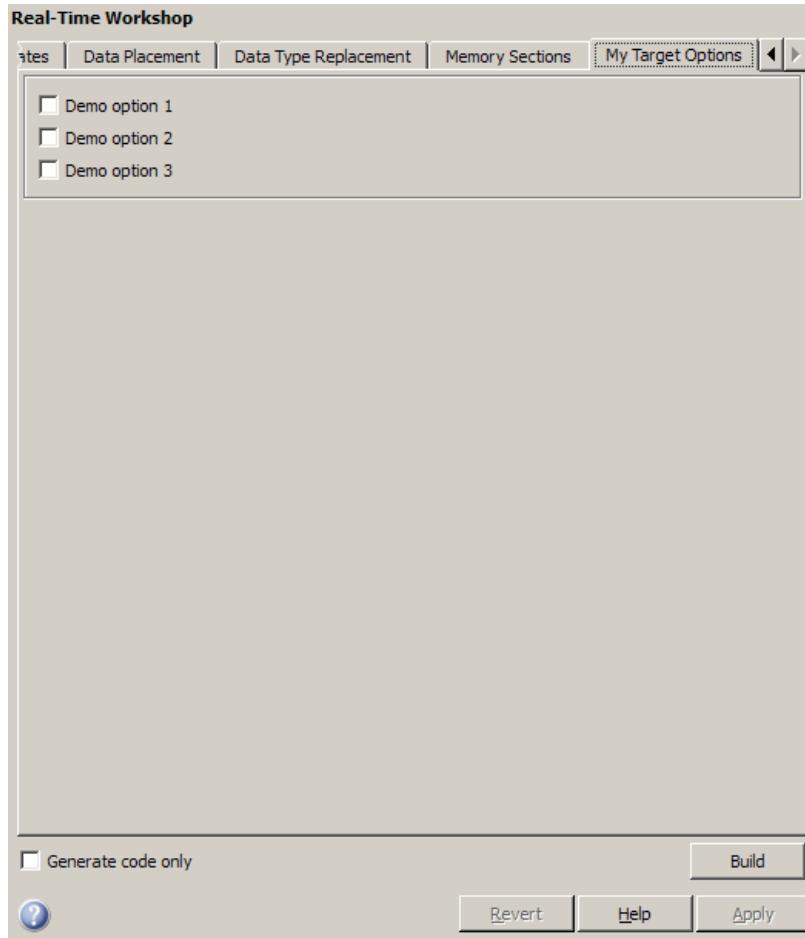
- Setting up target directories and modifying the MATLAB path.

- 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` on page 21-73).



Target-Specific Options for my_ert_target

`my_ert_target` supports a 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 Real-Time Workshop product. If you use a different compiler, you can set up LCC temporarily as your default compiler by typing the MATLAB command

```
mex -setup
```

Follow the prompts and select LCC.

Note On UNIX¹⁷ systems, make sure that you have a C compiler installed. You can then do this exercise, substituting appropriate UNIX folder syntax.

You can test `my_ert_target` with any model that is compatible with the ERT target. (See “Choosing and Configuring an Embedded Real-Time Target” in the Real-Time Workshop Embedded Coder documentation.) Generated programs operate identically to ERT generated programs.

However, to simplify the testing of your target, we recommend testing with `targetmodel.mdl`, a very simple fixed-step model (see “Create Test Model and S-Function” on page 21-82). The S-Function block in `targetmodel.mdl` uses the source code from the `timestwo` example, and generates fully inlined code. See the Simulink Writing S-Functions document and the *Real-Time Workshop Target Language Compiler* document for a complete discussion of the `timestwo` example S-function.

Creating Target Directories

In this section, you create directories to store the target files and add them to the MATLAB path, following the recommended conventions (see “Folder and File Naming Conventions” on page 21-11). You also create a folder to store the test model, S-function, and generated code.

This example assumes that your target and model directories are located within the folder `c:/work`. Note that your target and model directories should not be located anywhere in the MATLAB folder tree (that is, in or under the `matlabroot` folder).

To create the necessary directories 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
```

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

- 2** Within the target root folder, create a subfolder to store your target files.

```
mkdir my_ert_target/my_ert_target
```

- 3** Add these directories 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 Directories” on page 21-74.

```
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 21-41 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 21-55. 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 21-45.

To set the suffix to a string appropriate to 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 “`rtwgenseettings` Structure” on page 21-45. Add the following code to the `rtwgenSettings` definition:

```
rtwgenseettings.DerivedFrom = 'ert.tlc';
rtwgenseettings.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 %
%-----%

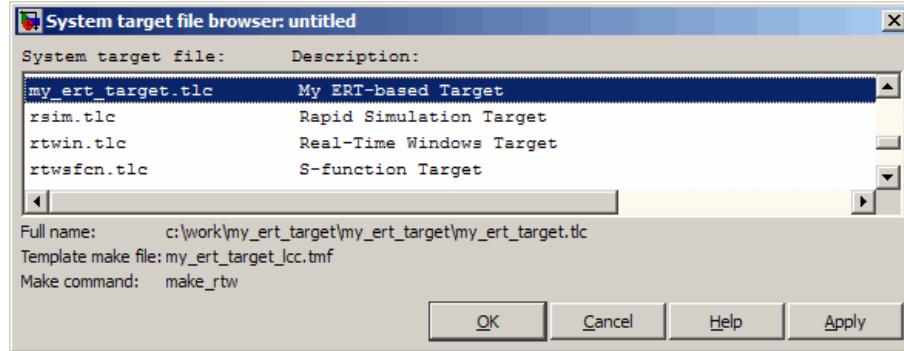
rtwgensettings.BuildDirSuffix = '_my_ert_target_rtw';
rtwgensettings.DerivedFrom = 'ert.tlc';
rtwgensettings.Version = '1';

END_RTW_OPTIONS
%/
```

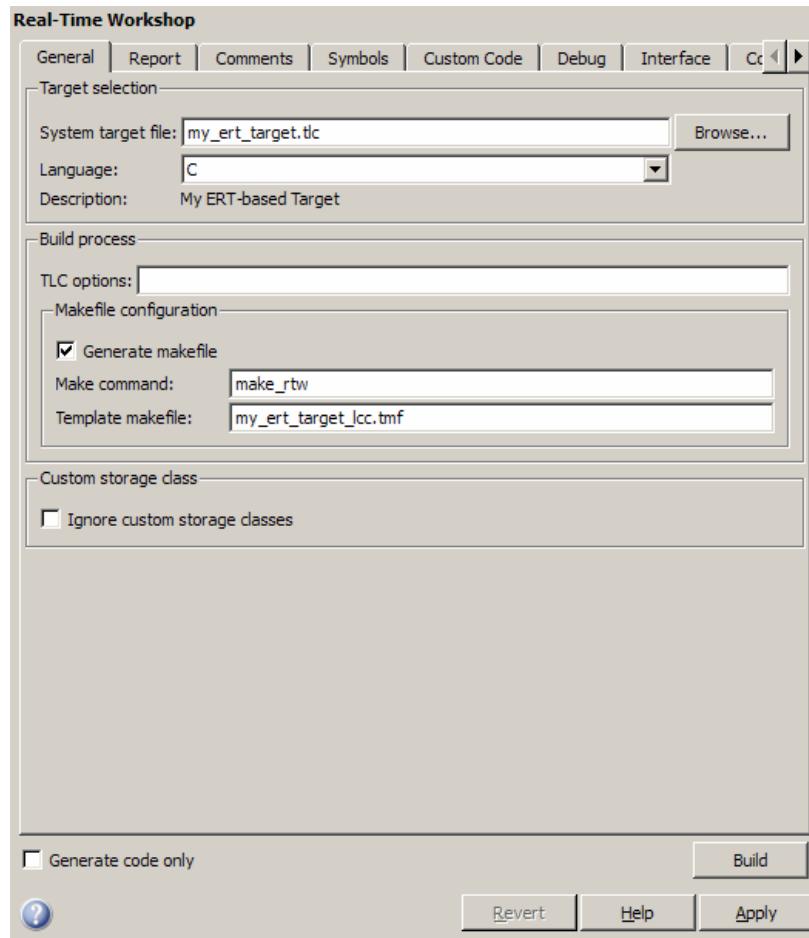
10 Save your changes to `my_ert_target.tlc` and close the file.

Viewing the STF. At this point, you can verify that the target inherits and displays ERT options correctly as follows:

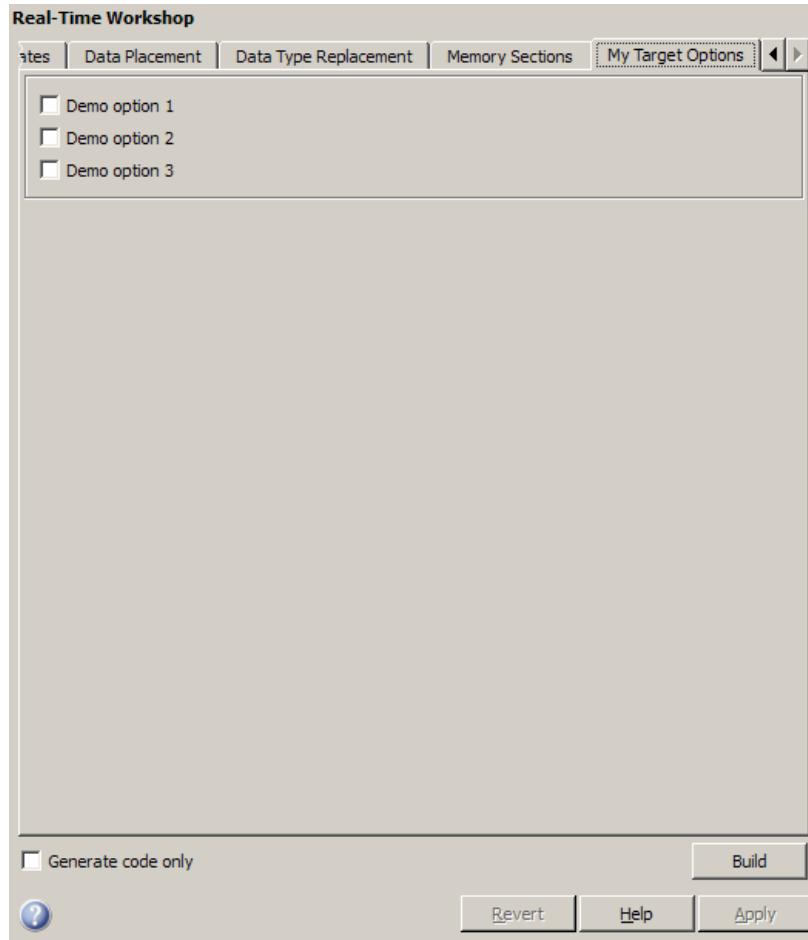
- 1** Create a new model.
- 2** Open the Model Explorer or the Configuration Parameters dialog box.
- 3** Select the **Real-Time Workshop** pane.
- 4** Click **Browse** to open the System Target File Browser.
- 5** In the Browser, scroll through the list of targets to find the new target, `my_ert_target.tlc`. (This step assumes that your MATLAB path contains `c:/work/my_ert_target/my_ert_target`, as previously set in “Creating Target Directories” on page 21-74.)
- 6** Select **My ERT-based Target** as shown below, and click **OK**.



- 7 The **Real-Time Workshop** 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 **Real-Time Workshop** pane and reopen the System Target File Browser.
- 10 Select the Real-Time Workshop 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 Directories” on page 21-74.

```
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 correct file reference 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.

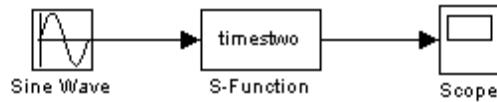
- 2** Copy the following C and TLC files for the `timestwo` S-function to your working folder:

- `matlabroot/simulink/src/timestwo.c`
- `matlabroot/toolbox/simulink/blocks/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.mdl`.



- 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 Model Explorer or the Configuration Parameters dialog box and select the **Real-Time Workshop** 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 **Real-Time Workshop** pane for `my_ert_target`.
- 5** Select the **Real-Time Workshop > 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 is successful, the MATLAB Command Window displays the message below.

```
### Created executable: ./targetmodel.exe
### Successful completion of Real-Time Workshop 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 a correct inlined output computation for the S-Function block in the model.

Customizing Template Makefiles

In this section...

- “Template Makefiles and Tokens” on page 21-86
- “Invoking the make Utility” on page 21-92
- “Structure of the Template Makefile” on page 21-93
- “Customizing and Creating Template Makefiles” on page 21-96

Template Makefiles and Tokens

- “Prerequisites” on page 21-86
- “Template Makefile Role In Makefile Creation” on page 21-86
- “Template Makefile Tokens” on page 21-87

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 Real-Time Workshop 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 **General** options section of the **Real-Time Workshop** 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` on page 21-87 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 appropriate macro definitions needed for the compilation of the files, for example, `-DINTEGER_CODE=1`.

Template Makefile Tokens Expanded by `make_rtw`

Token	Expansion
<code> >ALT_MATLAB_BIN< </code>	Alternate full pathname for the MATLAB executable; value is different than value for <code>MATLAB_BIN</code> token when the full pathname contains spaces.
<code> >ALT_MATLAB_ROOT< </code>	Alternate full pathname for the MATLAB installation; value is different than value for <code>MATLAB_ROOT</code> token when the full pathname contains spaces.
<code> >BUILDARGS< </code>	Options passed to <code>make_rtw</code> . This token is provided so that the contents of your <code>model.mk</code> file changes when you change the build arguments, thus forcing an update of all modules when your build options change.

Template Makefile Tokens Expanded by make_rtw (Continued)

Token	Expansion
>COMBINE_OUTPUT_UPDATE_FCNS<	True (1) when Single output/update function is selected, otherwise False (0). Used for the macro definition -DONESTEPFCN=1.
>COMPUTER<	Computer type. See the MATLAB computer command.
>EXT_MODE<	True (1) to enable generation of external mode support code, otherwise False (0).
>EXTMODE_TRANSPORT<	Index of transport mechanism (for example, tcpip , serial) for external mode.
>EXTMODE_STATIC<	True (1) if static memory allocation is selected for external mode. False (0) if dynamic memory allocation is selected.
>EXTMODE_STATIC_SIZE<	Size of static memory allocation buffer (if any) for external mode.
>GENERATE_ERT_S_FUNCTION<	True (1) when Create SIL block is selected, otherwise False (0). Used for control of the makefile target of the build.
>INCLUDE_MDL_TERMINATE_FCN<	True (1) when Terminate function required is selected, otherwise False (0). Used for the macro definition -DTERMFCN==1.

Template Makefile Tokens Expanded by make_rtw (Continued)

Token	Expansion
>INTEGER_CODE<	True (1) when Support floating-point numbers is not selected, otherwise False (0). INTEGER_CODE is a required macro definition when compiling the source code and is used when selecting precompiled libraries to link against.
>MAKEFILE_NAME<	<i>model.mk</i> — The name of the makefile that was created from the TMF.
>MAT_FILE<	True (1) when MAT-file logging is selected, otherwise False (0). MAT_FILE is a required macro definition when compiling the source code and also is used to include logging code in the build process.
>MATLAB_BIN<	Location of the MATLAB executable.
>MATLAB_ROOT<	Path to where MATLAB is installed.
>MEM_ALLOC<	Either RT_MALLOC or RT_STATIC. Indicates how memory is to be allocated.
>MEXEXT<	MEX-file extension. See the MATLAB <code>mexext</code> command.
>MODEL_MODULES<	Any additional generated source modules. For example, you can split a large model into two files, <i>model.c</i> and <i>model1.c</i> . In this case, this token expands to <i>model1.c</i> .
>MODEL_MODULES_OBJ<	Object filenames (.obj) corresponding to any additional generated source modules.

Template Makefile Tokens Expanded by make_rtw (Continued)

Token	Expansion
>MODEL_NAME<	Name of the Simulink block diagram currently being built.
>MULTITASKING<	True (1) if solver mode is multitasking, otherwise False (0).
>NCSTATES<	Number of continuous states.
>NUMST<	Number of sample times in the model.
>PORTABLE_WORDSIZES<	True (1) when Enable portable word sizes is selected, otherwise False (0).
>RELEASE_VERSION<	The MATLAB release version.
>S_FUNCTIONS<	List of noninlined S-function sources.
>S_FUNCTIONS_LIB<	List of S-function libraries available for linking.
>S_FUNCTIONS_OBJ<	Object (.obj) file list corresponding to noninlined S-function sources.
>SOLVER<	Solver source filename, for example, <code>ode3.c</code> .
>SOLVER_OBJ<	Solver object (.obj) filename, for example, <code>ode3.obj</code> .
>TARGET_LANG_EXT<	c when the Real-Time Workshop Language selection is C, cpp when the Language selection is C++. Used in the makefile to control the extension on generated source files.

Template Makefile Tokens Expanded by make_rtw (Continued)

Token	Expansion
>TGT_FCN_LIB<	Specifies compiler command line options. The line in the makefile is TGT_FCN_LIB = >TGT_FCN_LIB< . By default, the Real-Time Workshop build process expands the >TGT_FCN_LIB< token to match the setting of the Target function library option on the Real-Time Workshop/Interface pane of the Configuration Parameters dialog box. Possible values for this option include ANSI_C, C99 (ISO), GNU99 (GNU), and C++ (ISO). You can use this token in a makefile conditional statement to specify compiler options to be used. For example, if you set the token to C99 (ISO), the compiler might need an additional option set to support C99 library functions.
>TID01EQ<	True (1) if sampling rates of the continuous task and the first discrete task are equal, otherwise False (0).

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. Any 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'
```

Invoking the make Utility

- “make Command” on page 21-92
- “make Utility Versions” on page 21-93

make Command

After creating `model.mk` from your TMF, the Real-Time Workshop 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 21-93). You can specify additional options to `make` in the **Make command** field of the **Real-Time Workshop** pane. (See the sections “Specifying a Make Command” on page 25-12 and “Template Makefiles and Make Options” on page 18-32 in the Real-Time Workshop documentation.)

For example, specifying `OPT_OPTS=-02` in the **Make command** field causes `make_rtw` to generate the following `make` command.

```
makecommand -f model.mk OPT_OPTS=-02
```

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 Real-Time Workshop software provides the Free Software Foundation GNU `make` for both UNIX¹⁸ and PC platforms in platform-specific subdirectories under

```
matlabroot/bin
```

It is possible to use other versions of `make` with the Real-Time Workshop software, although GNU `make` is recommended. To ensure compatibility with the Real-Time Workshop software, make sure that your version of `make` supports the following command format.

```
makecommand -f model.mk
```

Structure of the Template Makefile

A TMF has four sections:

- The first section contains initial comments that describe what this makefile targets.
- The second section defines macros that tell `make_rtw` how to process the TMF. The macros are
 - `MAKECMD` — This is the command used to invoke the `make` utility. For example, if `MAKECMD = mymake`, then the `make` command invoked is
`mymake -f model.mk`
 - `HOST` — The target platform for this TMF is targeted for. This can be `HOST=PC`, `UNIX`, `computer_name` (see the MATLAB `computer` command), or `ANY`.

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

- **BUILD** — This tells `make_rtw` whether or not it should invoke `make` from the Real-Time Workshop build procedure. Specify `BUILD=yes` or `no`.
- **SYS_TARGET_FILE** — Name of the system target file or the value `all`. This is used for consistency checking by `make_rtw` to verify that the correct system target file was specified in the **Target selection** panel of the **Real-Time Workshop** pane of the Configuration Parameters dialog box. If you specify `all`, you can use the TMF with any system target file.
- **BUILD_SUCCESS** — An optional macro that specifies the build success string to be displayed on successful `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 Real-Time Workshop 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
```

- **BUILD_ERROR** — An 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** — add this command if you want the makefile output to be displayed always (regardless of the setting of the **Verbose build** option in the **Real-Time Workshop > Debugging** pane).

The following **DOWNLOAD** options apply only to the Wind River Tornado target:

- **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 successful 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]
```

- The third section defines the tokens make_rtw expands (see Template Makefile Tokens Expanded by make_rtw on page 21-87).
- The fourth 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 next figure shows the general structure of a Template Make File.

```
--- Section 1: Comments -----
#
# Description of target type and version of make for which
# this template makefile is intended.
# Also documents any optional build arguments.
--- Section 2: Macros read by make_rtw -----
#
# The following macros are read by the Real-Time Workshop build procedure:
#
# MAKECMD      - This is the command used to invoke the make utility.
# HOST         - Platform this template makefile is designed for
#                 (i.e., PC or UNIX)
# BUILD        - Invoke make from the Real-Time Workshop build procedure
#                 (yes/no)?
# SYS_TARGET_FILE - Name of system target file.

MAKECMD      = make
HOST         = UNIX
BUILD        = yes
SYS_TARGET_FILE = system.tlc
--- Section 3: Tokens expanded by make_rtw -----
#
MODEL        = |>MODEL_NAME<|
MODULES       = |>MODEL_MODULES<|
MAKEFILE       = |>MAKEFILE_NAME<|
MATLAB_ROOT    = |>MATLAB_ROOT<|
...
COMPUTER       = |>COMPUTER<|
BUILDARGS      = |>BUILDARGS<|
--- Section 4: Build rules -----
#
# The build rules are specific to your target and version of make.
```

} Comments

} make_rtw macros

} make_rtw tokens

} Build rules

Customizing and Creating Template Makefiles

- “Introduction” on page 21-97
- “Setting Up a Template Makefile” on page 21-97
- “Using Macros and Pattern Matching Expressions in a Template Makefile” on page 21-99
- “Using the rtwmakecfg.m API to Customize Generated Makefiles” on page 21-101

- “Supporting Continuous Time in Custom Targets” on page 21-107
- “Model Reference Considerations” on page 21-108
- “Generating Make Commands for Nondefault Compilers” on page 21-108

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 21-11 to understand the folder structure and MATLAB path requirements for custom targets.

Setting Up a Template Makefile

To customize or create a new TMF, you should copy an existing GRT or ERT TMF from one of the following locations:

```
matlabroot/rtw/c/grt  
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 appropriately (for example, `mytarget.tmf`) and modify it.

To ensure that the build process locates and selects your TMF correctly, you must provide information in the STF file header (see “System Target File Structure” on page 21-38). 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

If your target must support multiple development environments, you can specify a MATLABfile script that selects the correct TMF, based on user preferences. To do this, you must

- Create the script in your `mytarget/mytarget` folder. The naming convention for this file is `mytarget_default_tmf.m`.
- Specify this file in the TMF directive of the STF file header.

TMF: `mytarget_default_tmf`

The build process then invokes your `mytarget_default_tmf.m` file, which then selects the correct TMF, based on target preference settings. “`mytarget_default_tmf.m` Example Code” on page 21-98 illustrates this technique.

Another useful technique is to store a path to the user’s installed development environment in your target preferences. You can then locate the template makefiles under the appropriate tool folder. This allows several tool-specific template makefiles files to be located under the specific tool folder.

mytarget_default_tmf.m Example Code. The code example below implements a function, `mytarget_default_tmf`. The function loads target preferences into a structure from preferences data stored on disk. The code verifies that the target preferences information is consistent with the STF name, and extracts the associated TMF name. The TMF name is returned as the string `tmf`.

```
function [tmf,envVal] = mytarget_default_tmf
    try
        prefs = RTW.TargetPrefs.load('mytarget.prefs','structure');
    catch exception
        rethrow(exception);
    end

    % Get the desired MYTARGET implementation and ensure it is supported
    if ~isfield(prefs, 'Implementation')
        error('MYTARGET preferences not set correctly, update Target Preferences.');
    end
    imp = deblank(lower(prefs.Implementation));
    stfname = deblank(lower(get_param(bdroot, 'RTWSystemTargetFile')));

    if ~strcmp(imp, stfname, length(stfname) - length('.tlc'))
        msg = ['System Target filename: ', stfname,
               ' does not match Implementation specified in Target Preferences: ', imp];
    end
end
```

```

        error(msg);
end

% Return the desired template make file.
tmf = [imp, '.tmf'];

% This argument is unused
envVal = '';

```

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 needed to build or rebuild **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 UNIX¹⁹ 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 Real-Time Workshop 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 GNU²⁰ 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)
```

-
- 19. UNIX® is a registered trademark of The Open Group in the United States and other countries.
 - 20. GNU® is a registered trademark of the Free Software Foundation.

```

test: $(OBJS)
    cc -o $@ $(OBJS)

%.o : %.c
    cc -c $<

```

Note that the `$@` macro above is another special macro that equates to the name of the current dependency target, in this case `test`.

This example generates the list of objects (`OBJS`) from the list of sources (`SRCS`) by using the 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 21-101
- “Creating the `rtwmakecfg.m` Function” on page 21-102
- “Modifying the Template Makefile” on page 21-105

Overview. Real-Time Workshop TMFs provide tokens that let you add the following items to generated makefiles:

- Source directories
- Include directories
- 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 Real-Time Workshop software associates this file with your S-function based on its folder location. “Creating the `rtwmakecfg.m` Function” on page 21-102 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 21-105 discusses the required modifications.

After the TLC phase of the build process, when generating a makefile from the TMF, the Real-Time Workshop 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
<code>makeInfo.includePath</code>	A cell array that specifies additional include folder names, organized as a row vector. The Real-Time Workshop build process expands the folder names into include instructions in the generated makefile.
<code>makeInfo.sourcePath</code>	A cell array that specifies additional source folder names, organized as a row vector. The Real-Time Workshop build process expands the folder names into make rules in the generated makefile.
<code>makeInfo.sources</code>	A cell array that specifies additional source filenames (C or C++), organized as a row vector. The Real-Time Workshop 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 <code>sourcePath</code> field.

Field	Description
<code>makeInfo.linkLibsObjs</code>	A cell array that specifies additional, fully qualified paths to object or library files against which Real-Time Workshop generated code should link. The Real-Time Workshop 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 Real-Time Workshop 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 Real-Time Workshop build process.
<code>makeInfo.precompile</code>	A Boolean flag that indicates whether the libraries specified in the <code>rtwmakecfg.m</code> file exist in a specified location (<code>precompile==1</code>) or if the libraries need to be created in the build folder during the Real-Time Workshop build process (<code>precompile==0</code>).
<code>makeInfo.library</code>	A structure array that specifies additional run-time libraries and module objects, organized as a row vector. The Real-Time Workshop 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
<code>makeInfo.library(n).Name</code>	A character array that specifies the name of the library (without an extension).
<code>makeInfo.library(n).Location</code>	A character array that specifies the folder in which the library is located when precompiled. See the description of <code>makeInfo.precompile</code> in the preceding table for more information. A target can use the <code>TargetPreCompLibLocation</code> parameter to override this value. See “Specifying the Location of Precompiled Libraries” on page 27-8 in the Real-Time Workshop documentation for details.
<code>makeInfo.library(n).Modules</code>	A cell array that specifies the C or C++ source file base names (without an extension) that comprise the library. Do not include the file extension. The makefile appends the appropriate object extension.

Note The `makeInfo.library` field must fully specify each library and how to build it. The modules list in the `makeInfo.library(n).Modules` element cannot be empty. If you need to specify a link-only library, use the `makeInfo.linkLibsObjs` field instead.

Example:

```

disp(['Running rtwmakecfg from folder: ',pwd]);
makeInfo.includePath = { fullfile(pwd, 'somedir2') };
makeInfo.sourcePath = {fullfile(pwd, 'somedir2'), fullfile(pwd, 'somedir3')};
makeInfo.sources = { 'src1.c', 'src2.cpp' };
makeInfo.linkLibsObjs = { fullfile(pwd, 'somedir3', 'src3.object'),...
                        fullfile(pwd, 'somedir4', 'mylib.library')};
makeInfo.precompile = 1;
makeInfo.library(1).Name      = 'myprecompiledlib';
makeInfo.library(1).Location = fullfile(pwd,'somedir2','lib');
makeInfo.library(1).Modules   = {'srcfile1' 'srcfile2' 'srcfile3' };

```

Note If a path that you specify in the `rtwmakecfg.m` API contains spaces, the Real-Time Workshop 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 “Enabling the Real-Time Workshop Software to Build When Path Names Contain Spaces” on page 18-39 in the Real-Time Workshop 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 be appropriate for 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 “Controlling the Location and Naming of Libraries During the Build Process” on page 27-7 in the Real-Time Workshop 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 all 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, it is incorrect to 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 all instances of NUMST in the file by DISC_NUMST.

Model Reference Considerations

See “Supporting Model Referencing” on page 21-110 for important information on TMF modifications you may need to make to support the Real-Time Workshop model referencing features.

Generating Make Commands for Nondefault Compilers

Custom targets may need a target-specific hook file to generate an appropriate make command when a nondefault compiler is used. This file can be used to override the default Real-Time Workshop behavior for selecting the appropriate compiler tool to be used in the build process. See “STF_wrap_make_cmd_hook.m” on page 21-24 for further details.

Supporting Optional Features

In this section...
“Overview” on page 21-109
“Supporting Model Referencing” on page 21-110
“Supporting Compiler Optimization Level Control” on page 21-122
“Supporting firstTime Argument Control” on page 21-123
“Supporting C Function Prototype Control” on page 21-126
“Supporting C++ Encapsulation Interface Control” on page 21-128

Overview

This chapter describes how to configure a custom embedded target to support any of 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 inclusion of the <code>firstTime</code> argument in the <code>model_initialize</code> function generated for a Simulink model.	ERTFirstTimeCompliant (ERT only)
Controlling the C function prototypes of initialize and step functions that are generated for a Simulink model	ModelStepFunctionPrototypeControlCompliant (ERT only)
Generating and configuring C++ encapsulation interfaces to model code	CPPClassGenCompliant (ERT only)

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 any of 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 `s1ConfigUIGetVal`, `s1ConfigUISetEnabled`, and `s1ConfigUISetVal` for controlling custom target configuration options from a user-written `SelectCallback` function. (For function descriptions and examples, see “System Target File Callback Interface” in the Real-Time Workshop Embedded Coder reference documentation.)

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 `firstTime` argument control, C function prototype control, or C++ encapsulation 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.

Supporting Model Referencing

- “Overview” on page 21-111
- “Declaring Model Referencing Compliance” on page 21-112
- “Providing Model Referencing Support in the TMF” on page 21-112
- “Controlling Configuration Option Value Agreement” on page 21-116

- “Supporting the Shared Utilities Folder” on page 21-117
- “Preventing Resource Conflicts (Optional)” on page 21-121

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 21-112.
- The template makefile (TMF) must define some entities that support model referencing, as described in “Providing Model Referencing Support in the TMF” on page 21-112.
- The TMF must support using the Shared Utilities folder, as described in “Supporting the Shared Utilities Folder” on page 21-117.
- To optionally configure a target to support parallel builds for large model reference hierarchies (see “Reducing Build Time for Referenced Models” on page 25-23 in the Real-Time Workshop documentation), the STF and TMF must be modified for parallel builds as described in “Declaring Model Referencing Compliance” on page 21-112 and “Providing Model Referencing Support in the TMF” on page 21-112.
- You can optionally define additional capabilities that support model referencing, as described in “Preventing Resource Conflicts (Optional)” on page 21-121.

See “Referencing a Model” for information about model referencing in Simulink models, and “Creating Model Components” on page 3-51 for information about model referencing in Real-Time Workshop 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 `rtwgensesttings` 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`:

```
rtwgensesttings.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 “Reducing Build Time for Referenced Models” on page 25-23 in the Real-Time Workshop 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 “System Target File Callback Interface” in the Real-Time Workshop Embedded Coder reference documentation.

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.

Token	Expands to
MODELREF_LINK_LIBS token for the top model	List of referenced model libraries that the top model links against.
MODELREF_LINK_RSPFILE token for the top model	Name of a response file that the top model links against. This token is valid only for build environments that support linker response files. For an example of its use, see <i>matlabroot/rtw/c/grt/grt_vc.tmf</i> .
MODELREF_INC_PATH token for the top model	Include path to the referenced models.
RELATIVE_PATH_TO_ANCHOR	Relative path, from the location of the generated makefile, to the MATLAB working folder.
MODELREF_TARGET_TYPE	<p>Signifies the type of target being built. Possible values are</p> <ul style="list-style-type: none"> • NONE: Standalone model or top model referencing other models • RTW: Model reference Real-Time Workshop target build • SIM: Model reference simulation target build

If you are configuring your target to support parallel builds, as discussed in “Reducing Build Time for Referenced Models” on page 25-23 in the Real-Time Workshop 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 any 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 any existing final target rule). 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 submodels to compile files that reside in the MATLAB working folder (pwd).

```
% .o : $(RELATIVE_PATH_TO_ANCHOR) /%.c
$(CC) -c $(CFLAGS) $<
```

Controlling Configuration Option Value Agreement

By default, the value of any 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 “Customizing System Target Files” on page 21-37 for information about TLC-based system target files, and **rtwoptions** Structure Fields Summary on page 21-51 for a list of all **rtwoptions** fields.

Supporting the Shared Utilities Folder

- “Overview” on page 21-117
- “Implementing Shared Utilities Folder Support” on page 21-119

Overview. The makefile used by the Real-Time Workshop 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 **Utility function generation** options have the following requirements:

- **Auto**
 - Standalone model build — All files go to the build folder; no makefile updates needed.
 - Referenced model or top model build — Use shared utilities folder; makefile requires full model reference support.
- **Shared location**
 - Standalone model build — Use shared utility 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 (`s1prj/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 “Project Directory Structure for Model Reference Targets” on page 3-53 in the Real-Time Workshop 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. Note that support for the shared utilities folder is a necessary, but not sufficient, condition for supporting Model Reference builds. See the preceding sections of this chapter to learn about additional updates that are needed for supporting Model Reference builds.

The exact syntax of the changes can vary due to differences in the make utility and compiler/archiver tools used by your target. The examples below are based on the GNU²¹ make utility. You can find the following updated TMF examples for GNU and Microsoft Visual C++ make utilities in the GRT and ERT target directories:

- 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.

21. GNU® is a registered trademark of the Free Software Foundation.

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 directories for shared source files and the library compiled from the source files. In the current release, all 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 =
ifeq ($($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 all 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 UNIX²².

```
$(SHARED_LIB) : $(SHARED_OBJS)  
@echo "### Creating $@ "  
ar r $@ $(SHARED_OBJS)  
@echo "### Created $@ "
```

Note Depending on your make utility, you may be able to combine Steps 6 and 7 into one rule. For example, gmake (used with ert_unix.tmf) uses:

```
$(SHARED_LIB) : $(SHARED_SRC)  
@echo "### Creating $@ "  
cd $(SHARED_BIN_DIR); $(CC) -c $(CFLAGS) $(GCC_WALL_FLAG_MAX) $(notdir $?)  
ar ruvs $@ $(SHARED_OBJS)  
@echo "### $@ Created "
```

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)
```

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

```

$(LD) $(LDFLAGS) -o $@ $(LINK_OBJS) $(LIBS)
$(SHARED_LIB) $(SYSLIBS)
@echo "### Created executable: $(MODEL)"

```

- 9 Remove any explicit reference 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 the needed information to your target compilation environment.

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(codeGen modelName, '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
```

Supporting Compiler Optimization Level Control

- “Overview” on page 21-122
- “Declaring Compiler Optimization Level Control Compliance” on page 21-123
- “Providing Compiler Optimization Level Control Support in the Target Makefile” on page 21-123

Overview

This chapter 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 **Real-Time Workshop** 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 Real-Time Workshop 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 21-123.
- 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 21-123.

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 `rtwgensesttings` 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`:

```
rtwgensesttings.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 “System Target File Callback Interface” in the Real-Time Workshop Embedded Coder reference documentation.

When the `CompOptLevelCompliant` target configuration parameter is set to `on`, the **Compiler optimization level** parameter is displayed in the **Real-Time Workshop** 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 The MathWorks as a model for making the modifications.

Supporting firstTime Argument Control

- “Overview” on page 21-124

- “Declaring `firstTime` Argument Control Compliance” on page 21-124
- “Providing `firstTime` Argument Control Support in the Custom Static Main Program” on page 21-125

Overview

This chapter describes how to configure a custom embedded target to support `firstTime` argument control. Without the described modifications, you will not be able to use the `IncludeERTFirstTime` model configuration parameter to control inclusion of the `firstTime` argument in the `model_initialize` function generated for your model. (For more information about `firstTime` argument control, see `model_initialize` in the Real-Time Workshop Embedded Coder reference documentation.)

The requirements for supporting `firstTime` argument control are as follows:

- The target must be derived from the ERT target.
- The system target file (STF) must declare `firstTime` argument control compliance, as described in “Declaring `firstTime` Argument Control Compliance” on page 21-124.
- If your target uses a custom static main program, the main program must handle the inclusion and suppression of the `firstTime` argument for a given model, as described in “Providing `firstTime` Argument Control Support in the Custom Static Main Program” on page 21-125.

Declaring `firstTime` Argument Control Compliance

To declare `firstTime` argument control compliance for your target, you must implement a callback function that sets the `ERTFirstTimeCompliant` flag, and then install the callback function in the `SelectCallback` field of the `rtwgenseettings` 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`:

```
rtwgenseettings.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 `ERTFirstTimeCompliant` flag as follows:

```
slConfigUISetVal(hDlg, hSrc, 'ERTFirstTimeCompliant', 'on');
slConfigUISetEnabled(hDlg, hSrc, 'ERTFirstTimeCompliant', false);
```

For more information about the STF callback API, see “System Target File Callback Interface” in the Real-Time Workshop Embedded Coder reference documentation.

When the `ERTFirstTimeCompliant` target configuration parameter is set to `on`, you can use the `IncludeERTFirstTime` model configuration parameter to control inclusion of the `firstTime` argument in the `model_initialize` function generated for your model.

Note If the `ERTFirstTimeCompliant` parameter is set to `off` for your selected target, you cannot control inclusion of the `firstTime` argument in the `model_initialize` function. If you attempt to include or suppress the `firstTime` argument using the `IncludeERTFirstTime` model configuration parameter, Real-Time Workshop Embedded Coder software ignores the request, and your generated `model_initialize` function will not reflect the requested change.

Providing `firstTime` Argument Control Support in the Custom Static Main Program

If your target uses a custom static main program, you must update the static main program to handle the inclusion and suppression of the `firstTime` argument for a given model. One way to do this is to

- 1 Make sure the target TLC file assigns 1 to `AutoBuildProcedure` when using a static main program. For example,

```
%assign AutoBuildProcedure = !GenerateSampleERTMain
```

- 2** In the generated header file `autobuild.h`, the macro `INCLUDE_FIRST_TIME_ARG` will be set to 0 if the `IncludeERTFirstTime` parameter is set to off or 1 if the parameter is set to on.
- 3** Inside the static main program, make sure to `#include autobuild.h` and then conditionally compile declarations and calls to the `model_initialize` function, based on the value of the `INCLUDE_FIRST_TIME_ARG` macro.

Supporting C Function Prototype Control

- “Overview” on page 21-126
- “Declaring C Function Prototype Control Compliance” on page 21-127
- “Providing C Function Prototype Control Support in the Custom Static Main Program” on page 21-127

Overview

This chapter 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 “Controlling Generation of Function Prototypes” in the Real-Time Workshop 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 21-127.
- 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 21-127.

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 `rtwgensesttings` 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`:

```
rtwgensesttings.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 “System Target File Callback Interface” in the Real-Time Workshop Embedded Coder reference documentation.

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 appropriate 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 appropriately.

Supporting C++ Encapsulation Interface Control

- “Overview” on page 21-128
- “Declaring C++ Encapsulation Interface Control Compliance” on page 21-128

Overview

This chapter describes how to configure a custom embedded target to support C++ encapsulation interface control. Without the described modifications, you will not be able to use the **C++ (Encapsulated)** language option and the **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Configuration Parameters dialog box to generate and configure C++ encapsulation interfaces to model code. (For more information about C++ encapsulation interface control, see “Controlling Generation of Encapsulated C++ Model Interfaces” in the Real-Time Workshop Embedded Coder documentation.)

The requirements for supporting C++ encapsulation interface control are as follows:

- The target must be derived from the ERT target.
- The system target file (STF) must declare C++ encapsulation interface control compliance, as described in “Declaring C++ Encapsulation Interface Control Compliance” on page 21-128.

Declaring C++ Encapsulation Interface Control Compliance

To declare C++ encapsulation 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

`rtwgensesttings` 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`:

```
rtwgensesttings.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 “System Target File Callback Interface” in the Real-Time Workshop Embedded Coder reference documentation.

When the `CPPClassGenCompliant` target configuration parameter is set to `on`, you can use the `C++ (Encapsulated)` language option and the **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Configuration Parameters dialog box to generate and configure C++ encapsulation interfaces to model code.

Note Selecting the `C++ (Encapsulated)` language value 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++ encapsulation interface configured model step method appropriately, and demonstrates how the generated code can be deployed.

Interfacing to Development Tools

In this section...

“Introduction” on page 21-130

“Makefile Approach” on page 21-131

“Interfacing to an Integrated Development Environment” on page 21-131

Introduction

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 Real-Time Workshop 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 Real-Time Workshop Embedded Coder software) to your requirements. This approach is discussed in “Makefile Approach” on page 21-131.

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 “Interfacing to an Integrated Development Environment” on page 21-131.

Makefile Approach

A template makefile provides information about your model and your development system. The Real-Time Workshop build process uses this information to create an appropriate makefile (.mk file) to build an executable program. The Real-Time Workshop 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 “Customizing Template Makefiles” on page 21-86.

The following sections of this document supplement the basic template makefile information in the Real-Time Workshop documentation:

- “Supporting Multiple Development Environments” on page 21-62
- “Supplying Development Environment Information to Your Template Makefile” on page 21-34
- “mytarget_default_tmf.m” on page 21-22

Interfacing to an Integrated Development Environment

- “Introduction” on page 21-132
- “Generating a CPP_REQ_DEFINES Header File” on page 21-132
- “Interfacing to the Freescale CodeWarrior IDE” on page 21-133

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 Real-Time Workshop 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 21-86).

The Real-Time Workshop 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_reqDefines.tlc, provides an example. The code generates a C header file, cpp_reqDefines.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_reqDefines_h.tlc
%openfile CPP_DEFINES = "cpp_reqDefines.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 21-133. The solution described is applicable to any ASCII project file format.
- 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 21-138. The solution described is applicable to any IDE 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 all the necessary 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 virtually any ASCII file format (see "Generating a CPP_REQ_DEFINES Header File" on page 21-132).

To illustrate the basic concept, consider a hypothetical XML file exported from a CodeWarrior stationery project. The following is a partial listing:

```
<target>
  <settings>
    ...
    <\settings>
      <file><name>foo.c<\name>
      <\file>
      ...
      <file><name>foobar.c<\name>
      <\file>
      <fileref><name>foo.c<\name>
      <\fileref>
      ...
      <fileref><name>foobar.c<\name>
      <\fileref>
  <\target>
```

Insert this XML code into an `%openfile/%closefile` block within a TLC file, `test.tlc`, as shown below.

```

%% test.tlc
%% This code will generate a file model_project.xml,
%% where model is the generating model name specified in
%% the CompiledModel.Name field of the model.rtw file.
%openfile XMLFileContents = %<CompiledModel.Name>_project.xml
<target>
    <settings>
        ...
        <\settings>
        <file><name>%<CompiledModel.Name>.c<\name>
        <\file>
        ...
        <file><name>foobar.c<\name>
        <\file>
        <fileref><name>%<CompiledModel.Name>.c<\name>
        <\fileref>
        ...
        <fileref><name>foobar.c<\name>
        <\fileref>
    <\target>
%closefile XMLFileContents
%selectfile NULL_FILE

```

Note the use of the TLC token `CompiledModel.Name`. The token is resolved and the resulting filename is included in the output stream. You can specify other information, such as paths and libraries, in the output stream by specifying other tokens defined in `model.rtw`. For example, `System.Name` may be defined as `<Root>/Subsystem1`.

Now suppose that `test.tlc` is invoked during a target's build process, where the generating model is `mymodel.md1`. 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

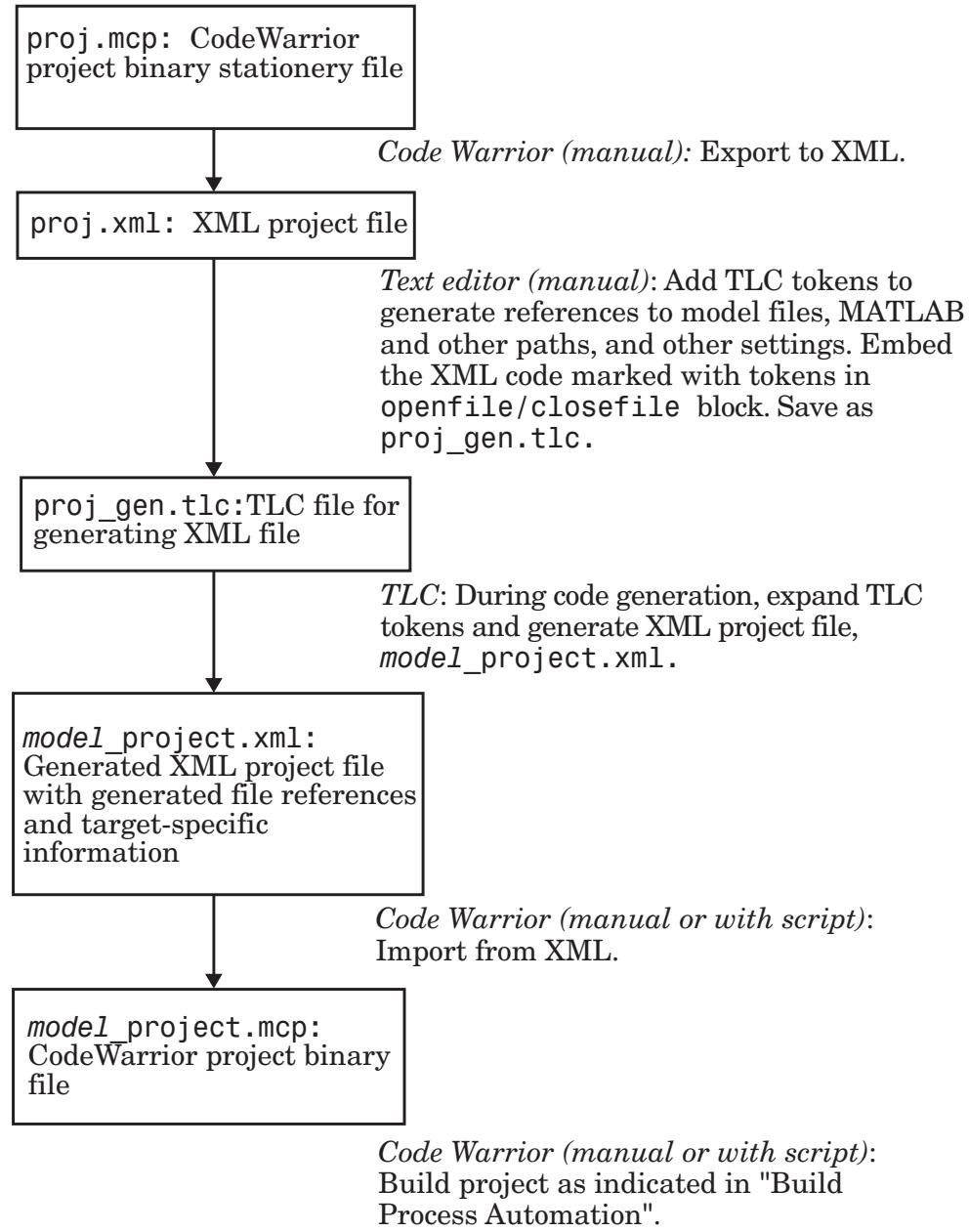
```
<file><name>%<CompiledModel.Name>.c<\name>
```

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 any 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 any other application that includes 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 the COM chapters of the *MATLAB External Interfaces* document.

For information about automation commands supported by the CodeWarrior IDE, see your CodeWarrior documentation.

COM automation is used by some embedded targets (for example, the Target Support Package product) 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 Target Support Package product initiates a build, the target

quickly invokes the necessary 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 21-23).

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. Another approach would be to let the user specify the location of project files with the target preferences.

```
%=====
% 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 correctly. 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 correctly. 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 correctly. 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 correctly. Verify COM access to
CodeWarrior outside of MATLAB.']);
    end
```

Device Drivers and Target Preferences

In this section...

[“Integrating Device Drivers” on page 21-142](#)

[“Using Target Preferences” on page 21-142](#)

Integrating 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 drivers functions into Simulink models by using the Legacy Code Tool. When you use the Real-Time Workshop product to generate code from a model, the Legacy Code Tool can insert an appropriate call to your C function into the generated code (for details, see “Automating Generation of Files for Fully Inlined S-Functions Using Legacy Code Tool” on page 32-25).

The Target Support Package product has a detailed demo showing how to use the Legacy Code Tool to integrate custom device driver code with the algorithmic code generated by Real-Time Workshop. If you have the Target Support Package product, see the demo “Custom Device Driver Integration via Legacy Code Tool”.

Using Target Preferences

You may want to associate certain types of data with the target. For example, an embedded target may offer users a choice of several supported development systems (cross-compilers, debuggers, and so on). To invoke the correct 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**

Generating Code and Building Executables

- Chapter 22, “Generating Code”
- Chapter 23, “Generating Reports for Code Reviews and Traceability Analysis”
- Chapter 24, “Reloading Generated Code”
- Chapter 25, “Building Executables”
- Chapter 26, “Optimizing Generated Code”
- Chapter 27, “Customizing Code Generation and the Build Process”



Generating Code

- “Generated Source Files and File Dependencies” on page 22-2
- “Understanding How Code Is Generated From a Model” on page 22-22

Generated Source Files and File Dependencies

In this section...

“Overview” on page 22-2

“Header Dependencies When Interfacing Legacy/Custom Code with Generated Code” on page 22-4

“Dependencies of the Generated Code” on page 22-14

“Specifying Include Paths in Real-Time Workshop Generated Source Files” on page 22-19

Overview

The Real-Time Workshop 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 Real-Time Workshop code generator handles most of the code formatting decisions (such as identifier construction and code packaging) in consistent ways.

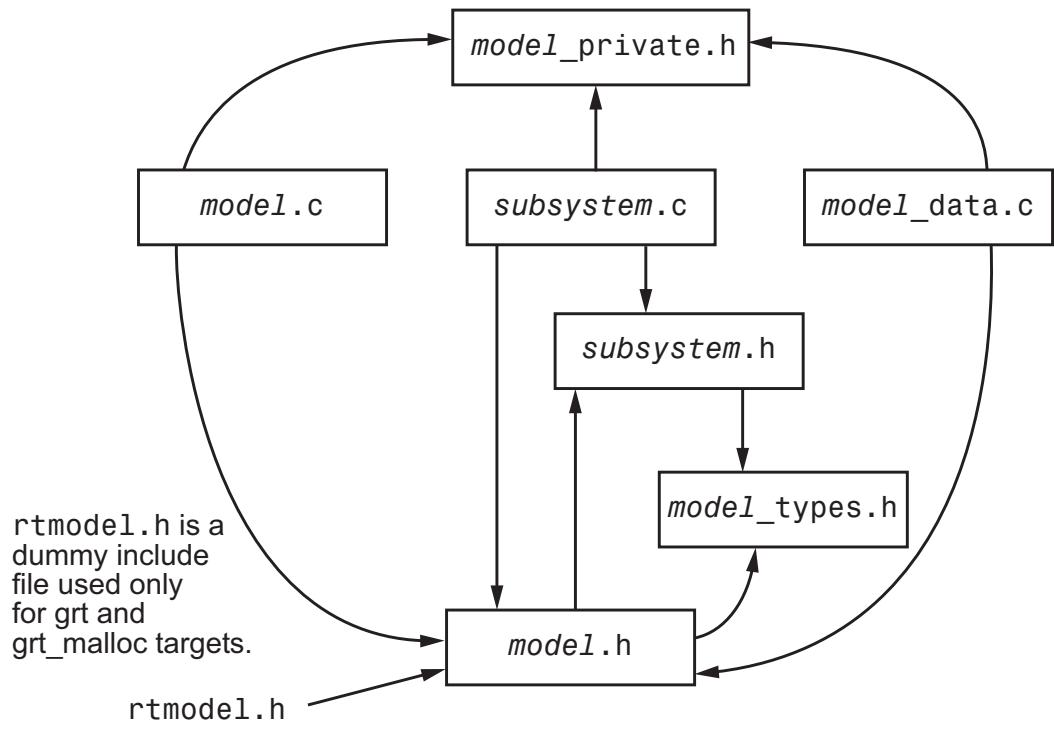
The source and make files created during the Real-Time Workshop build process are generated into your build directory, which is created or reused in your current directory. Some files are unconditionally generated, while the existence of others depend on target settings and options (for example, support files for C API or external mode). See “Files and Directories Created by the Build Process” on page 25-15 for descriptions of the generated files.

Note The file packaging of Real-Time Workshop Embedded Coder targets differs slightly from the file packaging described below. See “Code Modules” in the Real-Time Workshop 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 directory. Utility and model reference code located in a project directory might also be referenced by these files. See “Project Directory Structure for Model Reference Targets” on page 3-53 for details.

The figure shows that parent system header files (`model.h`) include all 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 all 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`).



Real-Time Workshop® Generated File Dependencies

Note In the preceding figure, files `model.h`, `model_private.h`, and `subsystem.h` also depend on the Real-Time Workshop header file `rtwtypes.h`. Targets that are not based on the ERT target can have additional dependencies on `tmwtypes.h` and `simstruct_types.h`.

Header Dependencies When Interfacing Legacy/Custom Code with Generated Code

You can incorporate legacy or custom code into a Real-Time Workshop build in any of several ways. One common approach is by creating S-functions. For details on this approach, see Integrating External Code With Generated C and C++ Code on page 1.

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 Real-Time Workshop product to make available the needed 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 <code>tmwtypes.h</code> and <code>simstruct_types.h</code> , both of which depend on <ul style="list-style-type: none"> • System headers <code>limits.h</code> and <code>float.h</code> • Real-Time Workshop headers: <code>rtw_matlogging.h</code>, <code>rtw_extmode.h</code>, <code>rtw_continuous.h</code>, and <code>rtw_solver.h</code>
ERT target and targets based on the ERT target	Is optimized, when possible, to include a minimum set of <code>#define</code> statements, enumerations, and so on; does not include <code>tmwtypes.h</code> and <code>simstruct_types.h</code>

The Real-Time Workshop build process generates the optimized version of `rtwtypes.h` for the ERT target when both of the following conditions exist:

- The **GRT compatible call interface** option on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box is cleared.
- The model contains no noninlined S-functions

You should always 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"

/* This ID is used to detect inclusion of an incompatible
 * rtwtypes.h
 */
#define RTWTYPES_ID_C08S16I32L32N32F1

#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
=====*/
/* This ID is used to detect inclusion of an incompatible rtwtypes.h */
#define RTWTYPES_ID_C08S16I32L32N32F1

/*=====
 * 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 Real-Time Workshop build process places `rtwtypes.h` in `s1prj/target/_sharedutils`; otherwise, it places `rtwtypes.h` in the standard build directory (`model_target_rtw`). See Chapter 39, “Setting Up Runtime Logging to MAT-Files” 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 Real-Time Workshop type names or other Real-Time Workshop definitions. A typical example is for files that declare variables using a Real-Time Workshop data type, for example, `uint32_T myvar;`.

A source file that is intended to be used by the Real-Time Workshop product and by a Simulink S-function can leverage the preprocessor macro `MATLAB_MEX_FILE`, which is defined by the `mex` function:

```

#ifndef MATLAB_MEX_FILE
#include "tmwtypes.h"
#else
#include "rtwtypes.h"
#endif

```

A source file meant to be used as the Real-Time Workshop `main.c` (or `.cpp`) file would also include `rtwtypes.h` without any 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.

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, as illustrated below, you should 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 Real-Time Workshop 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* or *main.cpp* is a static source file, a fixed header filename can be used, *rtmodel.h* for GRT or *autobuild.h* for ERT. These files include the *model.h* header file:

```
#include "model.h"      /* If main.c is generated */
```

or

```
#include "rtmodel.h" /* If static main.c is used with GRT */
```

or

```
#include "autobuild.h" /* If static main.c is used with ERT */
```

Other custom source files may also need to include *model.h* if there is a need 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 on page 22-9 and Real-Time Workshop® Header Files on page 22-11, 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 Real-Time Workshop interface option settings: <ul style="list-style-type: none"> • 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 an entry in the target function library (TFL) selected for the model <p>For more information about TFLs, see “Selecting and Viewing Target Function Libraries” on page 33-20 in this chapter, and the TFL chapter in the Real-Time Workshop Embedded Coder User’s Guide.</p>

System Header Files (Continued)

Header File	Purpose	GRT Targets	ERT Targets
<stddef.h>	Defines NULL	Included when the model contains a utility function that needs it	Included when the model contains a utility function that needs it
<stdio.h>	Provides file I/O functions	Included when the model includes a To File block	Included when the model includes a To File block, or you select Configuration Parameters > Real-Time Workshop > Interface > MAT-file logging . See “MAT-file logging”.
<stdlib.h>	Provides utility functions such as <code>div()</code> and <code>abs()</code>	Included when the model includes <ul style="list-style-type: none"> • A Stateflow chart • A Math Function block configured for <code>mod()</code> or <code>rem()</code>, which generate calls to <code>div()</code> 	Included when the model includes <ul style="list-style-type: none"> • A Stateflow chart and you select the Support floating-point numbers Real-Time Workshop interface configuration parameter • A Math Function block configured for <code>mod()</code> or <code>rem()</code>, which generate calls to <code>div()</code>
<string.h>	Provides memory functions such as <code>memset()</code> and <code>memcpy()</code>	Always included due to use of <code>memset()</code> in model initialization code	Included when block or model initialization code calls <code>memcpy()</code> or <code>memset()</code> <p>For a list of relevant blocks, enter <code>showblockdatatypetable</code> in the MATLAB Command Window and look for blocks with the N2 note.</p> <p>To omit calls to <code>memset()</code> from model initialization code, select the Remove root level</p>

System Header Files (Continued)

Header File	Purpose	GRT Targets	ERT Targets
			I/O zero initialization and Remove internal data zero initialization optimization configuration parameters.

Real-Time Workshop 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	Always included	Included when either of the following conditions exists: <ul style="list-style-type: none"> The model uses noninlined S-functions You select the Real-Time Workshop interface configuration parameter GRT compatible call interface
model_types.h	Defines model-specific data types	Always included	Always included
rt_logging.h	Supports MAT-file logging	Always included	Included when you select Configuration Parameters > Real-Time Workshop > Interface > MAT-file logging . See “MAT-file logging”.

Real-Time Workshop Header Files (Continued)

Header File	Purpose	GRT Targets	ERT Targets
<code>rt_nonfinite.h</code>	Provides support for nonfinite numbers in the generated code	Always included	Included when you select one of the following Real-Time Workshop interface configuration parameters: <ul style="list-style-type: none"> • MAT-file logging • Support non-finite numbers (and the generated code requires nonfinite numbers)
<code>rtw_continuous.h</code>	Supports continuous time	Always included by <code>simstruc_types.h</code>	Included when you select the Real-Time Workshop interface configuration parameter Support continuous time and <code>simstruc.h</code> is not already included
<code>rtw_extmode.h</code>	Supports external mode	Always included by <code>simstruc_types.h</code>	Included when you configure the model for external mode and <code>simstruc.h</code> is not already included
<code>rtw_matlogging.h</code>	Supports MAT-file logging	Included by <code>simstruc_types.h</code> and <code>rtw_logging.h</code>	Included by <code>rtw_logging.h</code>
<code>rtw_solver.h</code>	Supports continuous states	Always included by <code>simstruc_types.h</code>	Included when you select the Real-Time Workshop interface configuration parameter Support floating-point numbers and <code>simstruc.h</code> is not already included

Real-Time Workshop Header Files (Continued)

Header File	Purpose	GRT Targets	ERT Targets
<code>rtwtypes.h</code>	Defines Real-Time Workshop data types; generated file	Always included; use the complete version of the file, which includes <code>tmwtypes.h</code> and <code>simstruc_types.h</code> (see <code>simstruc_types.h</code> for dependencies)	Always included; use the complete or optimized version of the file as explained in “ <code>rtwtypes.h</code> ” on page 22-4
<code>simstruc.h</code>	Provides support for calling noninlined S-functions that use the Simstruct definition; also includes <code>limits.h</code> , <code>string.h</code> , <code>tmwtypes.h</code> , and <code>simstruc_types.h</code>	Always included	Included when either of the following conditions exists: <ul style="list-style-type: none"> The model uses noninlined S-functions You select the Real-Time Workshop interface configuration parameter GRT compatible call interface
<code>simstruc_types.h</code>	Provides definitions used by generated code and includes the header files <code>rtw_matlogging.h</code> , <code>rtw_extmode.h</code> , <code>rtw_continuous.h</code> , <code>rtw_solver.h</code> , and <code>sysran_types.h</code>	Always included with <code>rtwtypes.h</code>	Not included; <code>rtwtypes.h</code> contains needed definitions and <code>model.h</code> contains needed header files
<code>sysran_types.h</code>	Supports external mode	Always included by <code>simstruc_types.h</code>	Included when you configure the model for external mode and <code>simstruc.h</code> is not already included

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 Real-Time Workshop 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. All of these targets are typically makefile-based interfaces for which the Real-Time Workshop software provides a “Template MakeFile (TMF) to makefile” conversion capability. Part of this conversion process is to include in the generated makefile all of the source file, header file, and library file information needed (the dependencies) for a successful 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 any 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 directory for a standalone model or a subdirectory under the `slprj` directory 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. `main.c` (or `.cpp`) is a static (not generated) file (such as `grt_main.c` or `grt_main.cpp` for GRT-based targets), and is either a

static file (`ert_main.c` or `ert_main.cpp`) or is dynamically generated for ERT-based targets.

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 compiler's own run-time library, such as for functions in the ANSI²³ 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 directory for standalone models or in the `_sharedutils` directory under the `s1prj` directory for builds that involve model reference.
- Models with continuous time require solver source code files.
- Real-Time Workshop 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 Real-Time Workshop product provides several mechanisms for feeding file dependency information into the Real-Time Workshop build process. The

23. ANSI® is a registered trademark of the American National Standards Institute, Inc.

mechanisms available to you depend on whether your dependencies are block based or are model or target based.

For block dependencies, consider using

- S-functions and blocksets
 - Directories that contain S-function MEX-files used by a model are added to the header include path.
 - Makefile rules are created for these directories 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 “Using the **rtwmakecfg.m** API to Customize Generated Makefiles” on page 32-93 for more information.
- 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 **Real-Time Workshop/Custom Code** pane of the Configuration Parameters dialog box, which provides the ability to specify additional libraries, source files, and include directories.
- 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 Real-Time Workshop Embedded Coder

product also provides a TLC-based customization template capability for generating additional source files.

Makefile Considerations

As previously mentioned, Real-Time Workshop targets are typically makefile based and the Real-Time Workshop 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 “Customizing Template Makefiles” on page 21-86 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)
- Directories 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 there is a rule for that directory 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 is successful 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 Real-Time Workshop and blockset source and header files is provided below to assist in locating the dependencies.

Real-Time Workshop Static File Dependencies

Several locations in the MATLAB directory tree contain static file dependencies specific to the Real-Time Workshop product:

- *matlabroot/rtw/c/src/*

This directory has subdirectories 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 directory are included into the build process using in the SRC variables of the makefile.

- *matlabroot/rtw/extern/include/*.h*
- *matlabroot/simulink/include/*.h*

These directories contain additional header file dependencies such as `rtwtypes.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 necessary 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 Real-Time Workshop 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. The locations of the `rtwmakecfg.m` files for the blocksets are

- `matlabroot/commbblk/commbblkdemos/rtwmakecfg.m`
- `matlabroot/commbblk/commex/rtwmakecfg.m`
- `matlabroot/dspblk/dspmex/rtwmakecfg.m`
- `matlabroot/fuzzy/fuzzy/rtwmakecfg.m`
- `matlabroot/physmod/drive/drive/rtwmakecfg.m`
- `matlabroot/physmod/mec/mech/rtwmakecfg.m`
- `matlabroot/physmod/powersys/powersys/rtwmakecfg.m`

If the model being compiled uses one or more of these blocksets, you can determine directory and file dependency information from the respective `rtwmakecfg.m` file.

Specifying Include Paths in Real-Time Workshop 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 Real-Time Workshop 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:

- Real-Time Workshop generated code is compiled with a custom build process that requires an environment-specific set of `#include` statements.

In this scenario, the Real-Time Workshop code generator is likely invoked with the **Generate code only** check box selected. It may be appropriate to use 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 Real-Time Workshop build process.

In this case, compiler include paths (-I) can be provided to the Real-Time Workshop build process in several ways:

- The **Real-Time Workshop > 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 Real-Time Workshop build process. The include paths are propagated when the template makefile (TMF) is converted to the actual makefile.
- When building a custom Real-Time Workshop 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 directory in which the Real-Time Workshop build process should search for included files can be specified on the Real-Time Workshop `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 Real-Time Workshop build process to help ensure that the generated code remains portable and that compatibility problems with future versions are minimized.

Assume that additional header files are located at

```
c:\work\feature1\foo.h  
c:\work\feature2\bar.h
```

- A simple approach is to ensure all `#include` statements contain only the filename such as

```
#include "foo.h"
#include "bar.h"
```

Then, the include path passed to the compiler should contain all directories where 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 directory for these relative paths using an include path, for example,

```
#include "feature1\foo.h"
#include "feature2\bar.h"
```

Then specify the anchor directory (for example `\work`) to the compiler:

```
cc -Ic:\work ...
```

Directory Dependencies to Avoid

When using the Real-Time Workshop build process, avoid dependencies on its build and project directory structure, such as the `model_ert_rtw` build directory or the `s1prj` project directory. Thus, the `#include` statements should not just be relative to where the generated source file exists. For example, if your MATLAB current working directory is `c:\work`, a generated `model.c` source file would be generated into a subdirectory 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 Real-Time Workshop directory structure, you should instead use one of the approaches described above.

Understanding How Code Is Generated From a Model

In this section...

“Model Compilation” on page 22-22

“Code Generation” on page 22-22

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 the Simulink Writing S-Functions documentation).

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.

model.rtw files are similar in format to Simulink model (.mdl) files, but are used only for automated code generation. For a general description of the *model.rtw* file format, see the Target Language Compiler documentation.

Code Generation

The Real-Time Workshop 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 compiled 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** Compiles 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 any such file on your system prior to building.

- 3** Compiles 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 Real-Time Workshop software executes all 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

Generating Reports for Code Reviews and Traceability Analysis

- “Generating a Report” on page 23-2
- “Reviewing Generated Code” on page 23-3

Generating a Report

To generate an navigable summary of source files when the model is built, select the **Create code generation report** parameter on the Report pane. Selecting this parameter causes the Real-Time Workshop software to produce an HTML file for each generated source file, plus a summary and an index file, in a directory named `html` within the build directory. If you also select the **Launch report automatically** option (which is enabled by selecting **Create code generation report**), the HTML summary and index are automatically displayed. If you do not want to see the report at that time, clear this second check box. In either case, you can refer to HTML reports at any time. To review an existing HTML report, use any HTML browser to open the file `html/model_coden_rpt.html` within your build directory.

For more detail on report content, see “Viewing Generated Code in Generated HTML Reports” on page 23-3.

Reviewing Generated Code

You can view generated code in HTML reports or in the Model Explorer.

Viewing Generated Code in Generated HTML Reports

One way to view the generated code is to set the **Create code generation report** option on the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box. When set, this option generates a report that contains the following code generation details:

- A **Summary** section that lists version and date information, and a link to open configuration settings used for generating the code, including TLC options and Simulink model settings.
- A **Generated Files** section that contains a table of source code files generated from your model. You can view the source code in the MATLAB Help browser. When the Real-Time Workshop Embedded Coder product is installed, hyperlinks are placed within the source code that let you trace lines of code back to the blocks or subsystems from which the code was generated. Click the hyperlinks to highlight the relevant blocks or subsystems in a Simulink model window.

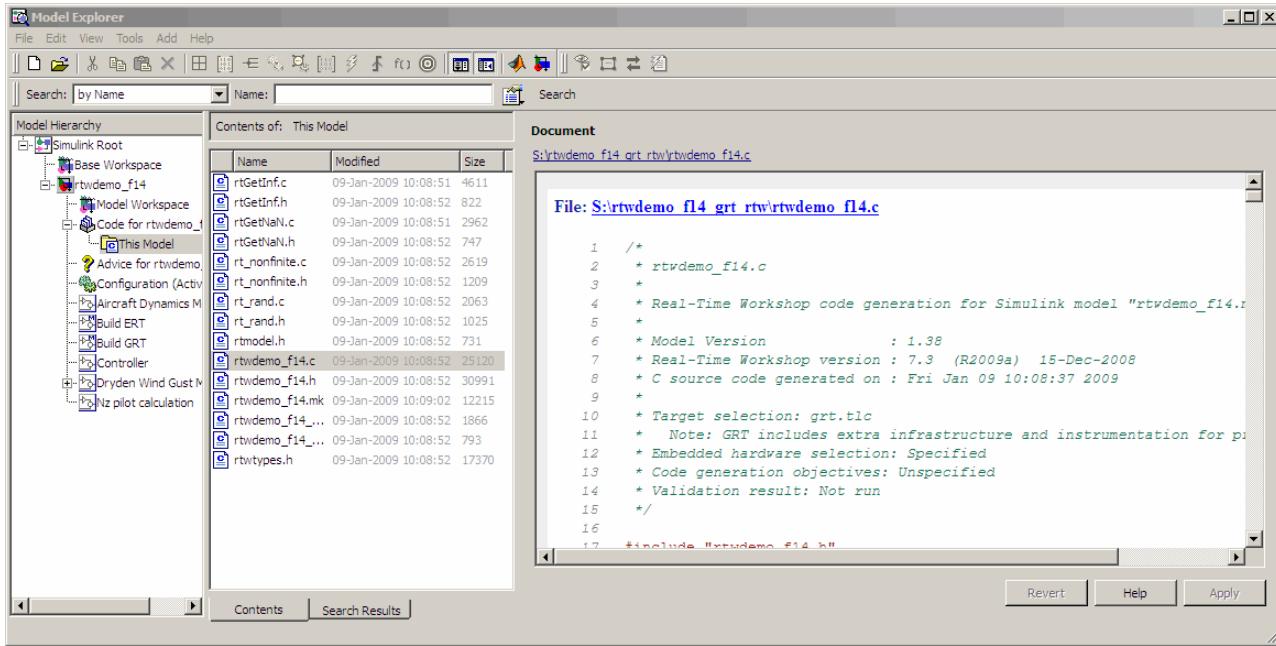
Note The report generated for various targets may vary slightly.

Viewing Generated Code in the Model Explorer Code Viewer

Another way to view the HTML source code report is to use the Code Viewer that is built into Model Explorer. You can browse 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 directory, 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 source code files

in the build directory of each model that is currently open. The next figure shows code for the `rtwdemo_f14` model.



In this example, the file `S:\rtwdemo_f14_grt_rtw\rtwdemo_f14.c` is being viewed. To view any 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. A hyperlink for the source file (not the HTML version you are looking at) being viewed sits above it. Clicking it opens that file in a text editing window where you can modify its contents. This is not something you typically do with generated source code, but in the event you have placed custom code files in the build directory, you can edit them as well in this fashion. You can also take advantage of your editor's features such as multipane display or custom syntax coloring.

If an open model contains Model blocks, and if generated code for any of these models exists in the current `s1prj` directory, 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.

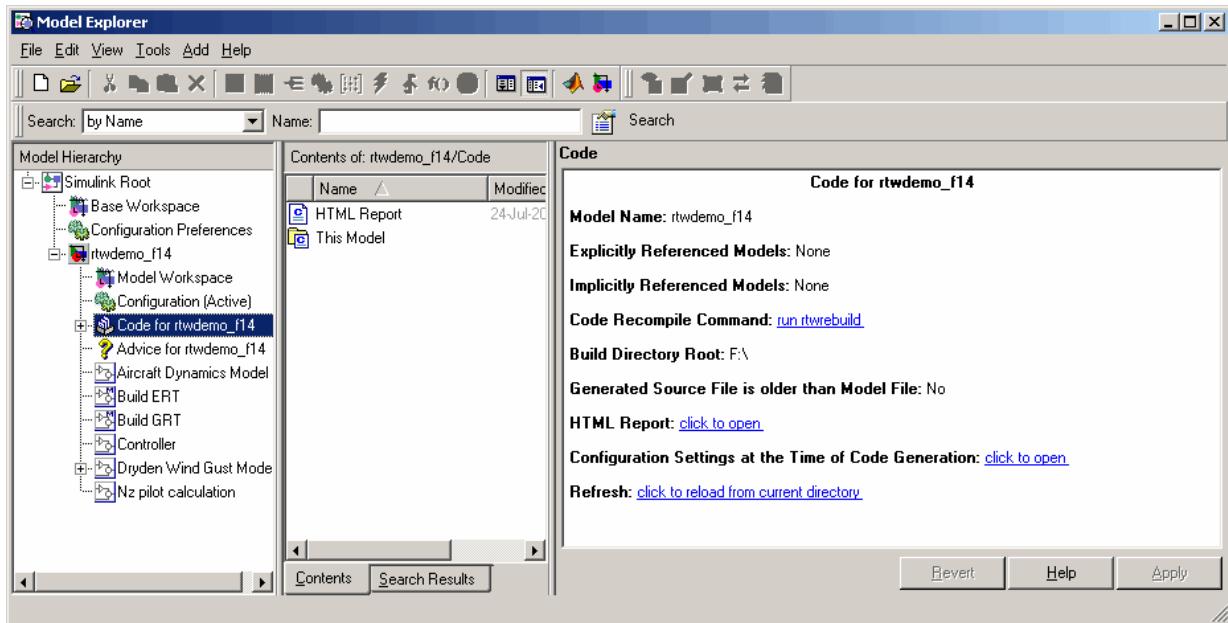
If the Real-Time Workshop software generates shared utility code for a model, a node named **Shared Code** appears directly under the **This Model** node. It collects any source files that exist in the appropriate `./slprj/target/_sharedutils` subdirectory.

Note Currently, you cannot use the **Search** tool built into Model Explorer's 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 you can use to search for strings in the currently displayed file. You can also search for text in the HTML report window, and can open any of the files in the editor.

Reloading 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 Real-Time Workshop software reloads the code for the model from the build directory.

24 Reloading Generated Code

24-2

Building Executables

- “Choosing and Configuring a Compiler” on page 25-2
- “Configuring the Build Process” on page 25-10
- “Initiating the Build Process” on page 25-14
- “Files and Directories Created by the Build Process” on page 25-15
- “Rebuilding a Model” on page 25-22
- “Reducing Build Time for Referenced Models” on page 25-23
- “Understanding the Build Process” on page 25-25
- “Relocating Code to Another Development Environment” on page 25-29

Choosing and Configuring a Compiler

In this section...

- “Compilers and the Build Process” on page 25-2
- “The Real-Time Workshop Product and ANSI²⁹ C/C++ Compliance” on page 25-3
- “Support for C and C++ Code Generation” on page 25-4
- “Support for International (Non-US-ASCII) Characters” on page 25-5
- “C++ Target Language Considerations” on page 25-8
- “Choosing and Configuring Your Compiler on a Microsoft Windows Platform” on page 25-8
- “Choosing and Configuring Your Compiler on The Open Group UNIX Platforms” on page 25-9
- “Including S-Function Source Code” on page 25-9

Compilers and the Build Process

The Real-Time Workshop build process depends upon the correct installation of one or more supported compilers. *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 Real-Time Workshop targets create an executable that runs on your workstation. When creating the executable, the Real-Time Workshop build process must be able to access an appropriate 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 template makefile. The template makefile determines which compiler runs, during the make phase of the build, to compile the generated code.

29. ANSI® is a registered trademark of the American National Standards Institute, Inc.

To determine which template makefiles are appropriate for your compiler and target, see Targets Available from the System Target File Browser on page 18-24.

For both Real-Time Workshop 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 Real-Time Workshop 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.

The Real-Time Workshop Product and ANSI²⁴ C/C++ Compliance

The Real-Time Workshop 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 Real-Time Workshop software from the following sources is ANSI C/C++ compliant:

- Simulink built-in block algorithmic code
- Real-Time Workshop and Real-Time Workshop Embedded Coder system level code (task ID [TID] checks, management, functions, and so on)
- Code from other blocksets, including the Simulink Fixed Point product, the Communications Blockset™ product, and so on
- Code from other code generators, such as the Stateflow® Coder™ product and Embedded MATLAB functions

Additionally, the Real-Time Workshop software can incorporate code from

24. ANSI is a registered trademark of the American National Standards Institute, Inc.

- 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 Real-Time Workshop 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.

Support for C and C++ Code Generation

Real-Time Workshop supports C and C++ code generation. The primary motivation for C++ support is to facilitate integration of generated code with legacy or custom user code written in C++. Consider the following as you choose a language for your generated code:

- Whether you need to configure Real-Time Workshop to use a specific compiler. This is required to generate C++ code on Windows. See “Choosing and Configuring a Compiler” on page 25-2.
- The language configuration setting for the model. See Chapter 19, “Selecting a Target Programming Language”.
- Whether you need to integrate legacy or custom code with generated code. For a summary of integration options, see “Integration Options” on page 29-2.
- Whether you need to integrate C and C++ code. If so, see Chapter 29, “About External Code Integration”.

Note You can mix C and C++ code when integrating Real-Time Workshop 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 wherever it is appropriate. For the details of the differing linkage conventions and how to apply `extern "C"`, refer to a C++ programming language reference book.

- “C++ Target Language Limitations” on page 25-5.

For a demo, enter `sfcndemo_cppcount` in the MATLAB Command Window.
 For a Stateflow example, enter `sf_cpp`.

C++ Target Language Limitations

- Real-Time Workshop does not support C++ code generation for the following:

SimDriveline™
 SimMechanics™
 SimPowerSystems™
 Target Support Package
 xPC Target

- The following Real-Time Workshop 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 the Model Explorer

These restrictions on specifying .cpp will be removed in a future release.

Support for International (Non-US-ASCII) Characters

Real-Time Workshop does not include non-US-ASCII characters in compilable portions of source code. However, Simulink, Stateflow, Real-Time Workshop, and Real-Time Workshop Embedded Coder do support non-US-ASCII characters in certain ways. When non-US-ASCII characters are encountered during code generation, they either become comments in the generated code or do not propagate into the generated source files. Sources of non-US-ASCII characters are described below:

- **Simulink Block Names:** The name of Simulink blocks are permitted to use non-US-ASCII character sets. The block name can be output in a comment above the generated code for that block when the **Simulink**

block / Stateflow object comments check box is selected. If Real-Time Workshop also uses the block name in the generated code as an identifier, the identifier's name will be changed to ensure only US-ASCII characters are present.

One exception to using non-US-ASCII characters in block names is for nonvirtual subsystems configured to use the subsystem name as either the function name or the filename. In this case, only US-ASCII characters can be used to name the subsystem.

- User comments on Stateflow diagrams: These comments can contain non-US-ASCII characters. They are written to the generated code when the **Include comments** check box is selected.
- Custom TLC files (.t1c): User-created Target Language Compiler files can have non-US-ASCII characters inside both TLC comments and in any code comments which are output. The Target Language Compiler does not support non-US-ASCII characters in TLC variable or function names.

Additional Support with Real-Time Workshop Embedded Coder

Users of Real-Time Workshop Embedded Coder have additional international character support:

- Simulink Block Description: Real-Time Workshop Embedded Coder propagates block descriptions entered from Simulink Block Parameter dialog boxes into the generated code as comments when the **Simulink block descriptions** check box on the **Real-Time Workshop/Comments** pane of the Configuration Parameters dialog box is selected. Non-US-ASCII characters are supported for these descriptions.
- Real-Time Workshop Embedded Coder code template file: Code Generation Template (.cgt) files provide customization capability for the generated code. Any output lines in the .cgt file which are C or C++ comments can contain non-US-ASCII characters, for example the file banner and file trailer sections; these comments are propagated to the generated code. However, although TLC comments in .cgt files can contain non-US-ASCII characters, these TLC comments are not propagated to the generated code.
- Stateflow object descriptions: Stateflow object descriptions can contain non-US-ASCII characters. The description will appear as a comment above the generated code for that chart when the **Stateflow object descriptions** check box is selected.

- Simulink Parameter Object Description: Simulink Parameter Object descriptions can contain non-US-ASCII characters. The description will appear as a comment above the generated code when the Simulink data object descriptions check box is selected.
- MPT Signal Object Description: MPT object descriptions can contain non-US-ASCII characters. The description will appear as a comment above the generated code when the Simulink data object descriptions check box is selected.

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 (.mdl) 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 with no 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 demo model *rtwdemo_international*. This demo model is set up to work around the character limitations described above. If you run this demo 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 description of `s1CharacterEncoding` in “Model Construction” in the Simulink documentation.

C++ Target Language Considerations

To use the C++ target language support, you might need to configure the Real-Time Workshop software to use the appropriate compiler. For example, on a Microsoft Windows platform the default compiler is the Lcc C compiler shipped with the MATLAB product, which does not support C++. If you do not configure the Real-Time Workshop software to use a C++ compiler before you specify C++ for code generation, the following build error message appears:

```
The specified Real-Time Workshop target is configured to generate
C++, but the C-only compiler, LCC, is the default compiler. To
specify a C++ compiler, enter 'mex -setup' at the command prompt.
To generate C code, click (Open) to open the Configuration
Parameters dialog and set the target language to C.
```

Choosing and Configuring Your Compiler on a Microsoft Windows Platform

On Windows platforms, you can use the Lcc C compiler shipped with the MATLAB product, or you can install and use one of the supported Windows compilers.

The Real-Time Workshop code generator will choose a compiler based on the template makefile (TMF) name specified on the **Real-Time Workshop** pane of the Configuration Parameters dialog box. The simplest approach is to let the code generator pick a compiler based on your default compiler, as set up using the `mex -setup` function. When you use this approach, you do not need to define compiler-specific environment variables, and the Real-Time Workshop code generator determines the location of the compiler using information from the `mexopts.bat` file located in the preferences directory (use the `prefdir` command to verify this location).

To use this approach, the TMF filename specified must be an MATLAB language file that returns default compiler information by using the `mexopts.bat` file. Most targets provided by the Real-Time Workshop product use this approach, as when `grt_default_tmf` or `ert_default_tmf` is specified as the TMF name.

Alternatively, the name provided for the TMF can be a compiler-specific template makefile, for example `grt_vc.tmf`, which designates the Microsoft Visual C++ compiler. When you provide a compiler-specific TMF filename, the Real-Time Workshop code generator uses the default `mexopts.bat` information to locate the compiler if `mex` has been set up for the same compiler as the specified TMF. If `mex` is not set up with a default compiler, or if it does not match the compiler specified by the TMF, then an environment variable must exist for the compiler specified by the TMF. The environment variable required depends on the compiler.

Choosing and Configuring Your Compiler on The Open Group UNIX Platforms

On a UNIX platform, the Real-Time Workshop build process uses the default compiler. For all operating systems except the Sun™ operating system, `cc` is the default compiler. On a Sun operating system, the default is `gcc`.

You should choose the UNIX template makefile that is appropriate to your target. For example, `grt_unix.tmf` is the correct template makefile to build a generic real-time program on a UNIX platform.

Including S-Function Source Code

When the Real-Time Workshop code generator builds models with S-functions, source code for the S-functions can be either in the current directory or in the same directory as their MEX-file. The code generator adds an include path to the generated makefiles whenever it finds a file named `sfuncname.h` in the same directory that the S-function MEX-file is in. This directory must be on the MATLAB path.

Similarly, the Real-Time Workshop code generator adds a rule for the directory when it finds a file `sfuncname.c` (or `.cpp`) in the same directory as the S-function MEX-file is in.

Configuring the Build Process

In this section...

- “Controlling Compiler Optimization Level and Specifying Custom Optimization Settings” on page 25-10
- “Specifying TLC Options” on page 25-11
- “Specifying Whether To Generate a Makefile” on page 25-11
- “Specifying a Make Command” on page 25-12
- “Specifying the Template Makefile” on page 25-12

Controlling Compiler Optimization Level and Specifying Custom Optimization Settings

To control compiler optimizations for your Real-Time Workshop makefile build at 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 Simulink GUI level, rather than editing compiler flags into template makefiles (TMFs) or supplying compiler flags to Real-Time Workshop 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 Real-Time Workshop 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” in the Real-Time Workshop reference.

Specifying TLC Options

You can enter Target Language Compiler (TLC) command line options in the **TLC options** edit field, for example

- `-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)

Specifying TLC options does not add any flags to the **Make command** field, as do some of the targets available in the System Target File Browser.

For additional information, see “Setting Target Language Compiler Options” on page 27-21 for details, as well as the Target Language Compiler documentation.

Specifying Whether To Generate a Makefile

The **Generate makefile** option specifies whether the Real-Time Workshop build process is to generate a makefile for a model. By default, the Real-Time Workshop 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 any post code generation build processing, using a user-defined command, as explained in “Customizing Post Code Generation Build Processing” on page 27-14.

Specifying a Make Command

A high-level MATLAB command, invoked when a build is initiated, controls the Real-Time Workshop build process. Each target has an associated make command. The **Make command** field displays this command.

Almost all 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 arguments in the **Make command** field. These arguments include compiler-specific options, include paths, and other parameters. When the build process invokes the `make` utility, these arguments are passed along in the `make` command line.

"Template Makefiles and Make Options" on page 18-32 lists the **Make command** arguments you can use with each supported compiler.

Specifying 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 an appropriate template makefile for your development environment. For example, in "Real-Time Workshop 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 18-32 gives a detailed description of the logic by which the Real-Time Workshop 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 make file 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, this is necessary if you have written your own template makefile for a custom target environment or you.

If you specify your own template makefile, be sure to include the filename extension. If you omit the extension, the Real-Time Workshop 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
```

Initiating 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 **Real-Time Workshop** pane of the Configuration Parameters dialog box and click **Build**.
- Press **Ctrl+B**.
- Select **Tools > Real-Time Workshop > Build Model**.
- Invoke the `rtwbuild` command from the MATLAB command line.
- Invoke the `slbuild` command from the MATLAB command line, using one of the following syntax options:

```
slbuild model
slbuild model 'buildtype'
slbuild('model')
slbuild('model' 'buildtype')
```

For `model`, specify the name of a model for which you want to build a stand-alone Real-Time Workshop target executable or a model reference target. The `buildtype` can be one of the following:

- `ModelReferenceSimTarget` builds a model reference simulation target
- `ModelReferenceRTWTTarget` builds a model reference simulation and Real-Time Workshop targets
- `ModelReferenceRTWTTargetOnly` builds a model reference Real-Time Workshop target

The command initiates the build process with the current model configuration settings. If the model has not been loaded by the Simulink product, `slbuild` loads it before initiating the build process.

For more information on model referencing, see “Creating Model Components” on page 3-51.

Files and Directories Created by the Build Process

The following sections discuss

- “Files Created During the Build Process” on page 25-15
- “Directories Used During the Build Process” on page 25-19

Files Created During the Build Process

This section lists *model.** files created during the code generation and build process for the GRT and GRT `malloc` targets when used with stand-alone models. Additional directories and files are created to support shared utilities and model references (see Chapter 3, “Architecture Considerations” in the Real-Time Workshop documentation).

The build process derives many of the files from the *model.mdl* file you create with Simulink. You can think of the *model.mdl* file as a very high-level programming language source file.

Note Files generated by the Real-Time Workshop Embedded Coder build process are packaged slightly differently. Depending on model architectures and code generation options, the Real-Time Workshop 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 Real-Time Workshop build process. This file is analogous to the object file created from a high-level language source program. By default, the Real-Time Workshop 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 *model.mdl* and is generated by the Target Language Compiler. This file contains

- Include files *model.h* and *model_private.h*
- All 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” on page 22-4 in the Real-Time Workshop documentation for more information. This header file contains

- Imported Simulink data symbols
- Imported Stateflow machine parented data
- Stateflow entry points
- Real-Time Workshop 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 might be needed by function declarations of reusable functions. *model_types.h* is included by all 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 any 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 any user-defined structure data types used by the model
- Constant parameters
- ***model.exe* (Microsoft Windows platforms) or *model* (UNIX platforms),** generated in the current directory, not in the build directory

Executable program file created under control of the `make` utility by your development system (unless you have explicitly specified that Real-Time Workshop generate code only and skip the rest of the build process)

- ***model.mk***

Customized makefile generated by the Real-Time Workshop build process. This file builds an executable program file.

- ***rtmodel.h***

Contains `#include` directives required by static main program modules such as *grt_main.c* and *grt_malloc_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***

For GRT targets, a header file that includes *simstruc_types.h*, which, in turn, includes *tmwtypes.h*. For Real-Time Workshop Embedded Coder

ERT targets, `rtwtypes.h` itself provides the necessary `defines`, `enums`, and so on, instead of including `tmwtypes.h` and `simstruc_types.h`. The `rtwtypes.h` file generated for ERT is an optimized (reduced) file based on the settings provided with the model that is being built. See “Header Dependencies When Interfacing Legacy/Custom Code with Generated Code” on page 22-4 in the Real-Time Workshop documentation for more information.

- `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 always 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 always generated for GRT-based targets and optionally generated for other targets.

- `rtw_proj.tmw`

Real-Time Workshop file generated for the `make` utility to use to determine when to rebuild objects when the name of the current Real-Time Workshop project changes

- `model.bat`

Windows batch file used to set up the appropriate compiler environment and invoke the `make` command

- `modelsources.txt`

List of additional sources that should be included in the compilation.

Optional files:

- `model_targ_data_map.m`

MATLAB language file used by external mode to initialize the external mode connection

- `model_dt.h`

C header file used for supporting external mode. Declares structures that contain data type and data type transition information for generated model data structures.

- *subsystem.c*

C source code for each noninlined nonvirtual subsystem or copy thereof when the subsystem is configured to place code in a separate file

- *subsystem.h*

C header file containing exported symbols for noninlined nonvirtual subsystems. Analogous to *model.h*.

- *model_capi.h, model_capi.c*

Header and sources file that contain data structures that describe the model's signals, states, and parameters without using external mode. See Chapter 35, "Interacting with Target Application Data Using the C API" in Real-Time Workshop User's Guide for more information.

- *rt_sfcn_helper.h, rt_sfcn_helper.c*

Header and source files providing functions needed 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 Real-Time Workshop software generates a set of HTML files (one for each source file plus a *model_contents.html* index file) in the *html* subdirectory within your build directory.

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" on page 22-2 in the Real-Time Workshop documentation.

Directories Used During the Build Process

The Real-Time Workshop build process places output files in three directories:

- The working directory

If you choose to generate an executable program file, the Real-Time Workshop build process writes the file *model.exe* (Windows) or *model* (UNIX) to your working directory.

- The build directory — *model_target_rtw*

A subdirectory within your working directory. The build directory 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 directory stores generated source code and all other files created during the build process (except the executable program file).

- Project directory — *s1prj*

A subdirectory within your working directory. When models referenced via Model blocks are built for simulation or Real-Time Workshop code generation, files are placed in *s1prj*. The Real-Time Workshop Embedded Coder configuration has an option that places generated shared code in *s1prj* without the use of model reference. Subdirectories in *s1prj* provide separate places for simulation code, some Real-Time Workshop code, utility code shared between models, and other files. Of particular importance to Real-Time Workshop users are:

- Header files from models referenced by this model
s1prj/target/model/referenced_model_includes
- Model reference Real-Time Workshop target files
s1prj/target/model
- MAT-files used during code generation of model reference Real-Time Workshop target and stand-alone code generation
s1prj/target/model/tmwinternal
- Shared (fixed-point) utilities
s1prj/target/_sharedutils

See “Working with Project Directories” for more information on organizing your files with respect to project directories.

The build directory always 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 directory might also include

- *model.rtw*
- Object files (.obj or .o)
- Code modules generated from subsystems
- HTML summary reports of files generated (in the `html` subdirectory)
- TLC profiler report files
- Block I/O, state, and parameter tuning information file (*model_capi.c*)
- Real-Time Workshop project (*model.tmw*) files

For additional information about using project directories, see “Project Directory Structure for Model Reference Targets” on page 3-53 and “Supporting Shared Utility Directories in the Build Process” on page 3-88 in the Real-Time Workshop documentation.

Rebuilding 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** In the **Model Hierarchy** pane, expand the node for the model of interest.
- 2** Click the **Code for *model*** node.
- 3** In the **Code** pane, click run `rtwrebuild`, listed to the right of the label **Code Recompile Command**.

Reducing Build Time for Referenced Models

You can accelerate code generation and compilation for models containing large model reference hierarchies by building referenced models in parallel whenever possible. For example, you can distribute code generation and compilation for each model block across the cores of a multicore host computer.

To take advantage of this feature, you must have the Parallel Computing Toolbox software licensed and installed in your development environment. If the Parallel Computing Toolbox software is available and started with the `matlabpool` function, the Real-Time Workshop build process builds referenced models in parallel whenever possible.

For example, to use parallel building for a large model reference hierarchy on a desktop machine with four cores perform the following steps.

- 1 In the MATLAB Command Window, enter

```
matlabpool 4
```

The Parallel Computing Toolbox software sets up a pool of four MATLAB workers, one for each core.

- 2 Optionally, inspect the model reference hierarchy to determine, based on model dependencies, which models will be built in parallel. In the Simulink model window, select **Tools > Model Dependencies > Model Dependency Viewer > Referenced Model Instances**.

A Model Dependency Viewer opens.

- 3 Build your model.

Messages in the MATLAB Command Window record when each parallel or serial build starts and finishes.

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 host machine attributes such as amount of RAM and number of cores.

Beyond performance benefits, the use of parallel builds for referenced models is transparent, with no differences in resulting code and directory structure between serial and parallel builds.

The following notes apply to using parallel builds for model reference hierarchies.

- Parallel builds of referenced models support only local MATLAB workers. They do not support remote workers in MATLAB Distributed Computing Server configurations.
- The host machine should have an appropriate amount of RAM available for supporting the number of local workers (MATLAB sessions) that you plan to use. For example, setting `matlabpool` to 4 results in five MATLAB sessions on your machine, each using approximately 120 MB of memory at startup.
- You must setup the same MATLAB environment in each MATLAB worker session as in the MATLAB client session. For example, each environment needs the same base workspace variables, MATLAB path settings, and so forth. You can do this using the `PreLoadFcn` callback from the top model. Since the top model is loaded with each MATLAB worker session, you can use its preload function for any MATLAB worker session setup.

For information on how to configure a custom embedded target to support parallel builds, see “Supporting Model Referencing” on page 21-110.

Understanding the Build Process

In this section...

- “Build Process Steps” on page 25-25
- “Customized Makefile Generation” on page 25-25
- “Executable Program Generation” on page 25-26

Build Process Steps

The Real-Time Workshop 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 **Real-Time Workshop** 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 Real-Time Workshop 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 22-22
- 2 “Code Generation” on page 22-22
- 3 “Customized Makefile Generation” on page 25-25
- 4 “Executable Program Generation” on page 25-26

For more information, see “Generating a Report” on page 23-2. You can also view an HTML report in **Model Explorer**.

Customized Makefile Generation

After generating the code, the Real-Time Workshop 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.

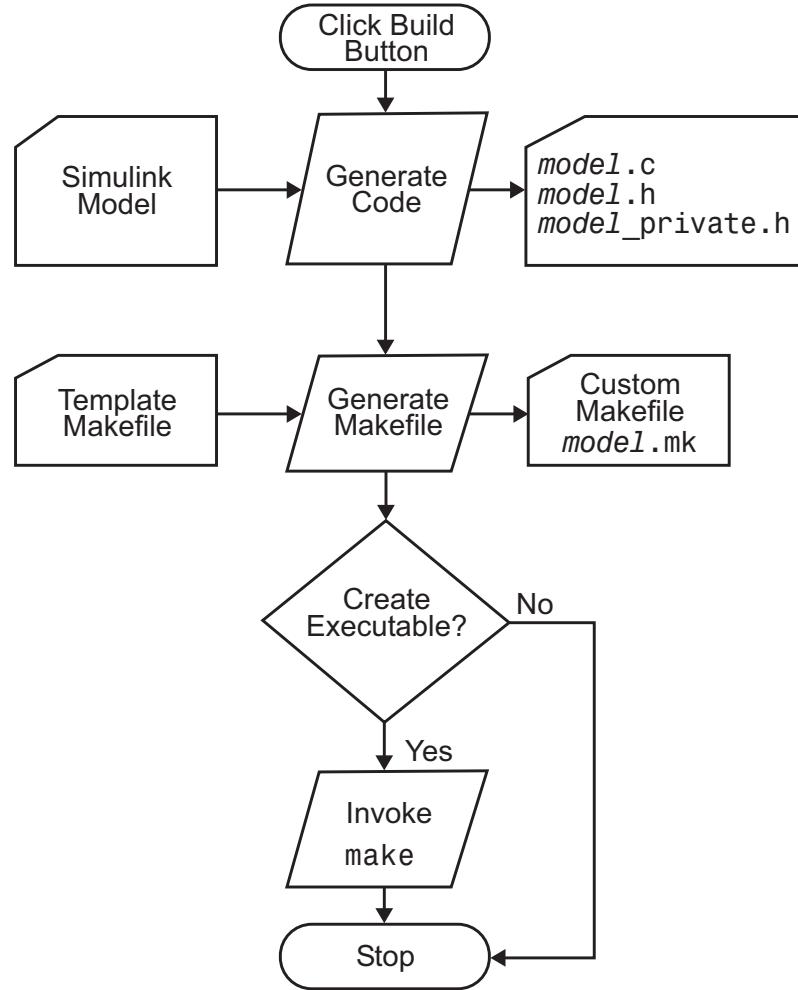
The Real-Time Workshop software creates *model.mk* from a system template file, *system.tmf* (where *system* stands for the selected target name). The system template makefile is designed for your target environment. You have the option of modifying the template makefile to specify compilers, compiler options, and additional information used during the creation of the executable.

The Real-Time Workshop 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 Real-Time Workshop software provides many system template makefiles, configured for specific target environments and development systems. “Selecting a Target” on page 18-29 in the Real-Time Workshop documentation lists all template makefiles that are bundled with the Real-Time Workshop 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 Real-Time Workshop software controls automatic program building.

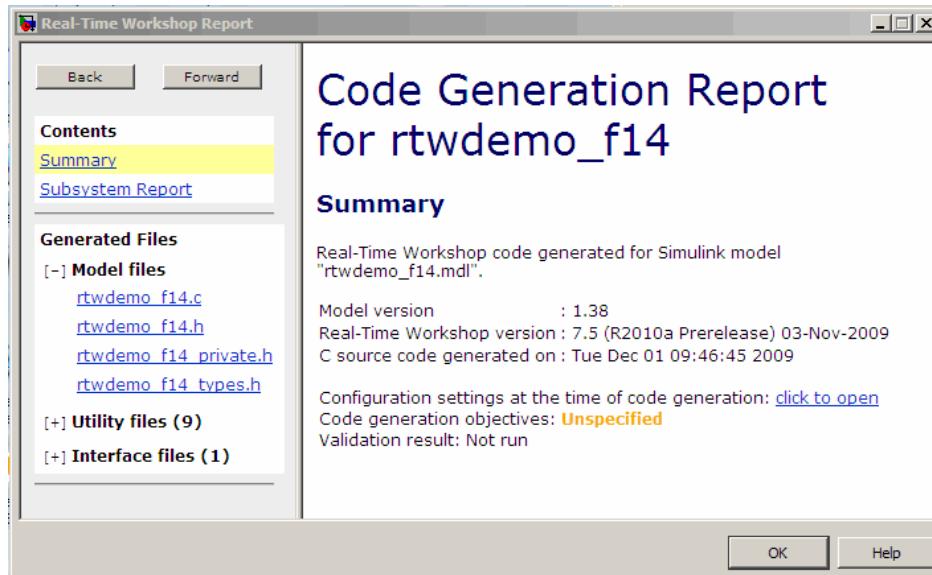


During the final stage of processing, the Real-Time Workshop build process invokes the generated makefile, `model.mk`, which in turn compiles and links the generated code. On PC platforms, a batch file is created to invoke the generated makefile. The batch file sets up the proper environment for invoking the `make` utility and related compiler tools. To avoid unnecessary recompilation of 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 **Real-Time Workshop** pane of the Configuration Parameters dialog box. You can then cross-compile your code and download it to your target hardware. Chapter 25, “Building Executables” in the Real-Time Workshop documentation discusses the options that control whether or not the build creates an executable image.

If you select **Create code generation report** on the **Real-Time Workshop > Report** pane, a navigable summary of source files is produced when the model is built. The report files occupy a directory called `html` within the build directory. The report contents vary depending on the target, but all reports feature links to generated source files. The following display shows an example of an HTML code generation report for a generic real-time (GRT) target.



Relocating Code to Another Development Environment

In this section...

- “Introduction” on page 25-29
- “Deciding on a Structure for the Zip File” on page 25-29
- “Deciding on a Name for the Zip File” on page 25-30
- “Packaging Model Code Files in a Zip File” on page 25-31
- “Inspecting the Generated Zip File” on page 25-32
- “Relocating and Unpacking the Zip File” on page 25-32
- “Code Packaging Example” on page 25-32
- “packNGo Function Limitations” on page 25-33

Introduction

If you need to relocate the static and generated code files for a model to another development environment, such as a secure system or an integrated development environment (IDE) that does not include MATLAB and Simulink products, use the Real-Time Workshop 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 all files needed to build an executable image. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility.

To relocate a model’s code files,

- 1** Decide on a structure of the zip file.
- 2** Decide on a name for the zip file.
- 3** Package the model code files in the zip file.
- 4** Relocate and unpack the zip file.

Deciding on 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 directory structure.

By default, the `packNGo` function packages all the necessary files in single, flat directory structure. This is the simplest approach and might be the optimal choice.

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 directory structure
The target development environment must maintain the directory 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` directory tree
- `sDirFiles.zip` — files in and under your build directory where you initiated the model's code generation
- `otherFiles.zip` — required files not in the `matlabroot` or start directory trees

Paths for the secondary zip files are relative to the root directory of the primary zip file, maintaining the source development directory structure.

Deciding on a Name for the Zip File

By default, the `packNGo` function names the primary zip file `model.zip`. You have the option of specifying a different name. If you specify a file name and omit the file type extension, the function appends `.zip` to the name you specify.

Packaging Model Code Files 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, use `set_param` to set the configuration parameter `PostCodeGenCommand` to an explicit call to the `packNGo` function before generating the model code. For example:

```
set_param(bdroot, 'PostCodeGenCommand', 'packNGo(buildInfo);');
```

This command instructs the Real-Time Workshop 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's code to disk and before generating a makefile.

- A build information object that you construct programmatically, as explained in “Customizing Post Code Generation Build Processing” on page 27-14.

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);
```

To change the default behavior of `packNGo`, see the following examples:

To...	Specify...
Change the structure of the file packaging to hierarchical	<code>packNGo(buildInfo, {'packType' 'hierarchical'});</code>
Rename the primary zip file	<code>packNGo(buildInfo, {'fileName' 'zippedsrcs'});</code>
Change the structure of the file packaging to hierarchical and rename the primary zip file	<code>packNGo(buildInfo, {'packType' 'hierarchical'... 'fileName' 'zippedsrcs'});</code>

Note The `packNGo` function potentially can modify the build information passed in the first `packNGo` argument. For example, as part of packaging model code, `packNGo` finds include files from all source and include paths recorded in the model's build information and adds them to the build information.

Inspecting the Generated Zip File

Inspect the resulting zip file in your working directory on the source system to verify that it is ready for relocation to the destination system. Depending on the zip tool 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 all files are preserved.

Relocating and Unpacking the Zip File

Relocate the resulting zip file to the destination development environment and unpack the file.

Code Packaging Example

The following example guides you through the steps for packaging code files generated for the demo model `rtwdemo_f14`.

- 1** Set your working directory to a writable directory.
- 2** Open the model `rtwdemo_f14` and save a copy to your working directory.
- 3** Enter the following command in the MATLAB Command Window:

```
set_param('rtwdemo_f14', 'PostCodeGenCommand',...
    'packNGo(buildInfo, {'packType' ''hierarchical'})');
```

Note that it is necessary to 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_f14.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 use of 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 may be included. For example, the function includes all S-function libraries in an all-or-nothing manner and all header files from every include directory, even if they are not used.

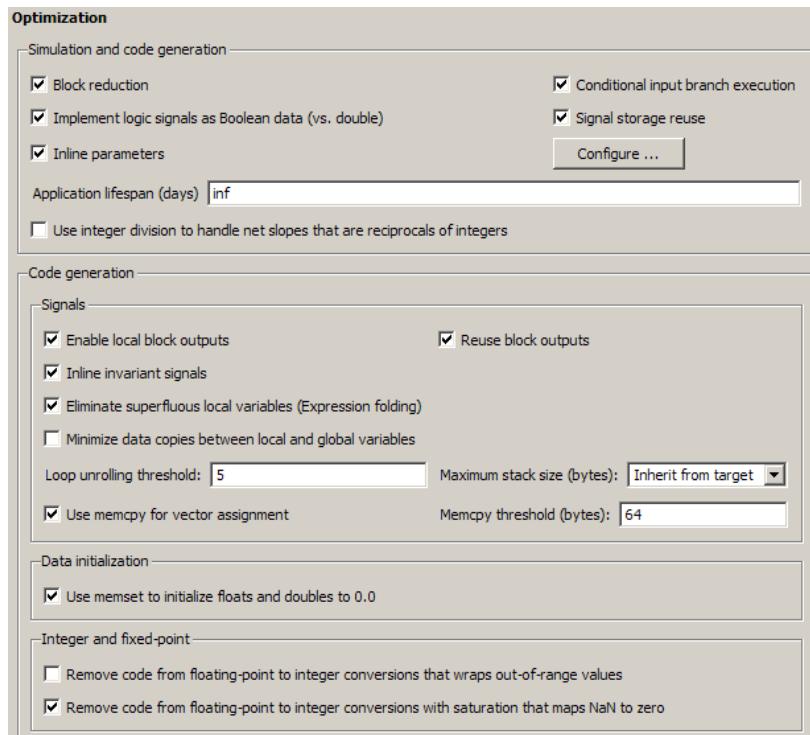
Optimizing Generated Code

You can optimize memory usage and performance of code generated from your model by the Real-Time Workshop product a number of ways. This chapter discusses optimization techniques that are common to all target configurations and code formats. For an overview of controlling optimization, see “Optimization Pane” in the Simulink documentation. For optimizations specific to a particular target configuration, see the chapters relevant to that target.

- “Optimization Parameters Overview” on page 26-2
- “Optimizing Models” on page 26-5
- “Minimizing Computations and Storage for Intermediate Results” on page 26-10
- “Block Diagram Performance Tuning” on page 26-15
- “Optimizing Signals” on page 26-39
- “Inlining Parameters” on page 26-43
- “Configuring a Loop Unrolling Threshold” on page 26-45
- “Optimizing Code Generated for Vector Assignments” on page 26-47
- “Controlling Memory Allocation for Time Counters” on page 26-51
- “Using Stack Space Allocation” on page 26-52
- “Optimizing Code Resulting from Floating-Point to Integer Conversions” on page 26-54
- “Improving Efficiency of Code That Uses Division by Constant Power of 2” on page 26-56
- “Optimization Dependencies” on page 26-63

Optimization Parameters Overview

Many options on the **Optimization** pane of the Configuration Parameters dialog box affect generated code. The next figure shows the default optimization settings, plus **Inline parameters** and **Inline invariant signals** (which are both off by default).



Some basic optimization suggestions are given below, cross-referenced to more extensive relevant discussions in the documentation.

- Selecting the **Signal storage reuse** check box directs the Real-Time Workshop code generator to store signals in reusable memory locations. It 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 (see below).

Disabling **Signal storage reuse** makes all block outputs global and unique, which in many cases significantly increases RAM and ROM usage. See “Reducing Memory Requirements for Signals” on page 26-39 for more details.

- 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. See “Inlining Parameters” on page 26-43 and “Using Model Arguments” in the Simulink documentation for more details.
- Selecting the **Enable local block outputs** check box declares block signals locally in functions instead of being declared globally (when possible). You must select **Signal storage reuse** to enable **Enable local block outputs**. See “Declaring Signals as Local Function Data” on page 26-40.
- Selecting the **Reuse block outputs** check box reduces stack size where signals are buffered in local variables. You must select **Signal storage reuse** to enable **Reuse block outputs**. See “Reusing Memory Allocated for Signals” on page 26-40.
- Selecting the **Inline invariant signals** check box makes the Real-Time Workshop code generator not generate code for blocks with a constant (invariant) sample time. You must select **Inline parameters** to enable **Inline invariant signals**. See “Inlining Invariant Signals” on page 26-41.
- 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. See “Minimizing Computations and Storage for Intermediate Results” on page 26-10.
- 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**. See “Reusing Memory Allocated for Signals” on page 26-40.
- Set an appropriate **Loop unrolling threshold**. The loop unrolling threshold determines when a wide signal should be wrapped into a `for` loop

and when it should be generated as a separate statement for each element of the signal. See “Configuring a Loop Unrolling Threshold” on page 26-45 for details on this feature.

- 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. See “Using Stack Space Allocation” on page 26-52 for more information.
- 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 an appropriate **Memcpy threshold (bytes)**. See “Optimizing Code Generated for Vector Assignments” on page 26-47 for details.

Optimizing Models

In this section...

“Getting Advice About Optimizing Models for Code Generation” on page 26-5

“Demos Illustrating Optimizations” on page 26-6

“Minimizing Memory Requirements for Parameters and Data During Code Generation” on page 26-6

“Other Optimization Tools and Techniques” on page 26-7

Getting Advice About Optimizing Models for Code Generation

Using the Model Advisor, you can quickly 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 Real-Time Workshop Embedded Coder software).

Use the following Model Advisor demos to investigate optimizing models for code generation using the Model Advisor:

- `rtdemo_advisor1`
- `rtdemo_advisor2`
- `rtdemo_advisor3`

Note Demo models `rtwdemo_advisor2` and `rtwdemo_advisor3` require Stateflow and Fixed-Point Toolbox™ software.

For more information on using the Model Advisor, see “Consulting the Model Advisor” in the Simulink documentation. For more information about the checks in the Model Advisor, see “Real-Time Workshop Checks” in the Real-Time Workshop documentation.

Demos Illustrating Optimizations

The `rtwdemos` demo suite includes a set of demonstration models that illustrate optimization settings and techniques. To access these demos, type

```
rtwdemos
```

or click the above command. The MATLAB Help browser opens the Real-Time Workshop demos page. Click **Optimizations** in the navigation pane. Use the listed demos to learn about the specific effects that optimization parameters and techniques have on models.

Minimizing Memory Requirements for Parameters and Data During Code Generation

When the Real-Time Workshop 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 performance 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 Real-Time Workshop 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 when it is needed 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 Real-Time Workshop 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.

Other Optimization Tools and Techniques

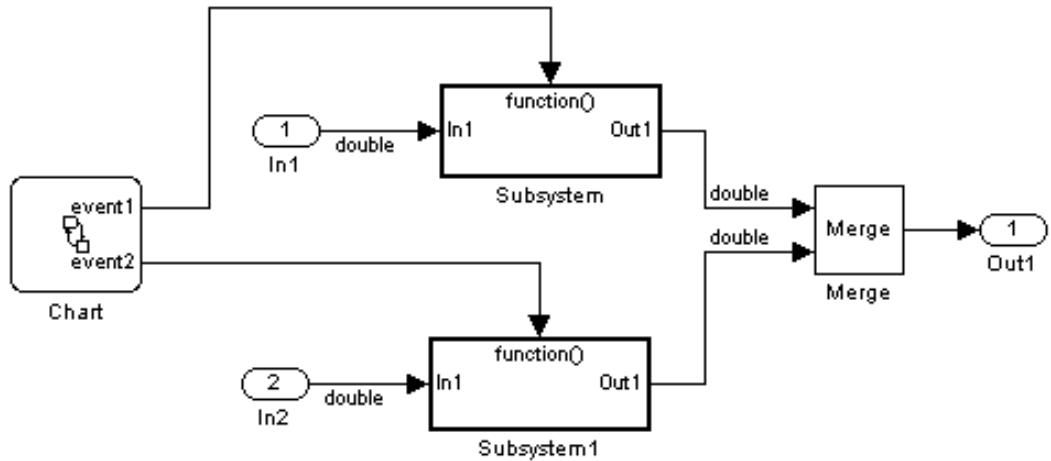
In addition to analyzing models with Model Advisor (see “Getting Advice About Optimizing Models for Code Generation” on page 26-5), you can use a variety of other tools and techniques that work with any code format. Here are some particularly useful ones:

- Run the `slupdate` command to automatically convert older models (saved by prior versions or by the current one) to use current features. For details about what `slupdate` does, type

```
help slupdate
```

- 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. See “Creating Component Object Libraries and Enhancing Simulation Performance” on page 3-92 and the Target Language Compiler documentation for more information on inlining S-functions.
- 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). See “Working with Data” in the Simulink documentation for more information on data types. For a block-by-block summary, click `showblockdatatypetable` or type the command in the MATLAB 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 all signals connected to any of the Merge blocks in the group.

If you are licensed to use the Real-Time Workshop Embedded Coder product, see also “Configuring Production Code Optimizations” and “Tips for Optimizing the Generated Code” in the Real-Time Workshop Embedded Coder documentation.

Minimizing Computations and Storage for Intermediate Results

In this section...

“Introduction” on page 26-10

“Expression Folding Example” on page 26-11

“Using and Configuring Expression Folding” on page 26-12

Introduction

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 Real-Time Workshop 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.

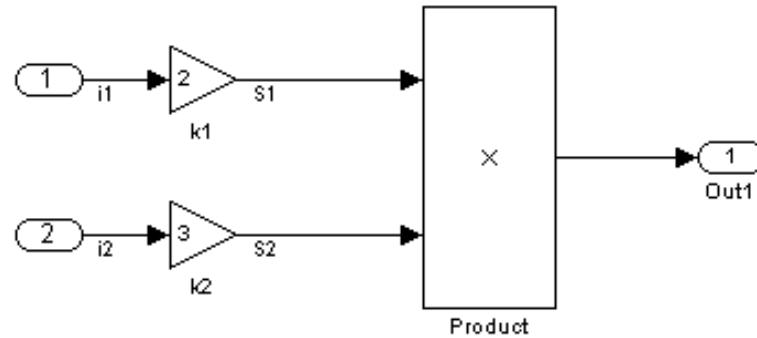
By default, expression folding is on. The Real-Time Workshop code generation options are configured to use expression folding wherever possible. Most Simulink blocks support expression folding.

You can also take advantage of expression folding in your own inlined S-function blocks. See “Writing S-Functions That Support Expression Folding” on page 32-57 for information on how to do this.

In the code generation examples that follow, the **Signal storage reuse** optimizations (**Enable local block outputs**, **Reuse block outputs**, **Eliminate superfluous local variables (Expression folding)** and **Minimize data copies between local and global variables**) are all turned on.

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;
}
```

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
```

Using and Configuring Expression Folding

The options described in this section let you control the operation of expression folding.

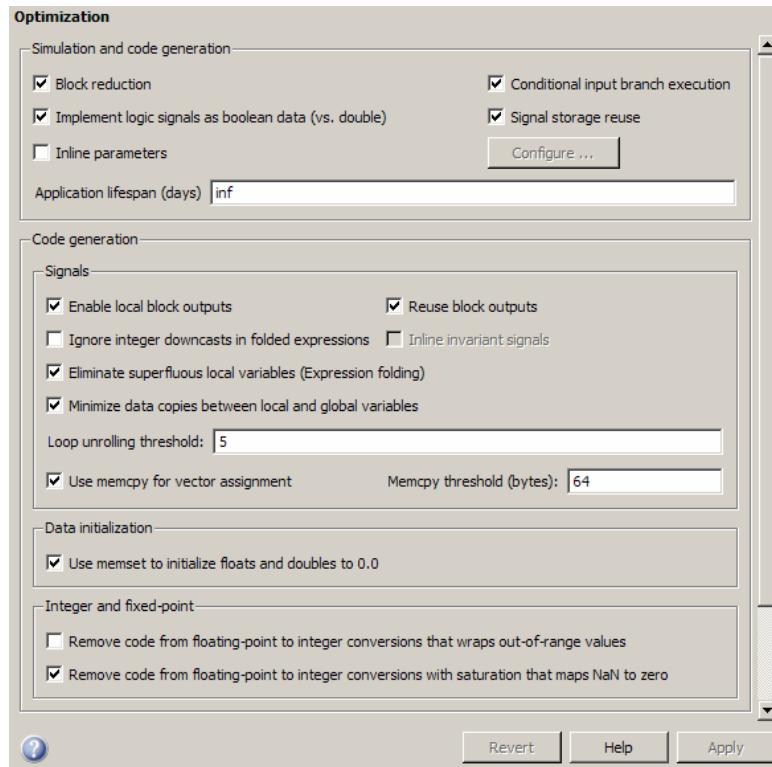
Enabling Expression Folding

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, you can ensure that expression folding is turned on as follows:

- 1** Open the Configuration Parameters dialog box and select the **Optimization** pane.
- 2** Select the **Signal storage reuse** check box.
- 3** Select the **Enable local block outputs** check box.
- 4** Select the **Reuse 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)**.

The **Optimization** pane appears in the next figure. All expression folding related options are selected, as shown.



7 Click **Apply**.

Block Diagram Performance Tuning

In this section...

- “Introduction” on page 26-15
- “Lookup Tables and Polynomials” on page 26-15
- “Reducing the Number of Blocks in a Model” on page 26-28
- “Optimizing Code for Switch Blocks” on page 26-32
- “Optimizing Data Type Usage” on page 26-33
- “Additional Integer and Fixed-Point Optimizations” on page 26-37

Introduction

Certain Simulink block constructs will run faster, or require less code or data memory, than other seemingly equivalent constructs. Knowing the tradeoffs between similar blocks and block parameter options enables you to create Simulink models that have intuitive diagrams, and to produce the tight code that you want using the Real-Time Workshop product. Many of the options and constructs discussed in this section improve the simulation speed of the model itself, even without code generation.

Lookup Tables and Polynomials

The Simulink product provides several blocks that allow approximation of functions. These include blocks that perform direct, interpolated, and cubic spline lookup table operations, and a polynomial evaluation block.

There are currently six different Simulink blocks that perform lookup table operations:

- Look-Up Table
- Look-Up Table (2-D)
- Look-Up Table (n-D)
- Direct Look-Up Table (n-D)
- PreLook-Up Index Search

- Interpolation (n-D) Using PreLook-Up Index Search

In addition, the Repeating Sequence block uses a lookup table operation, the output of which is a function of the real-time (or simulation-time) clock.

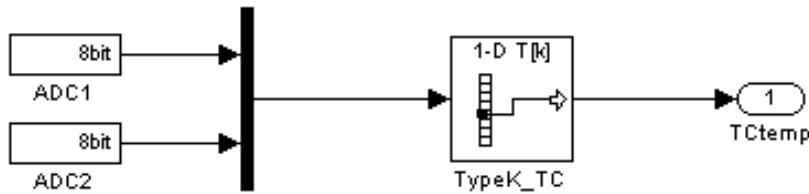
To get the most out of the following discussion, you should familiarize yourself with the features of these blocks, as discussed in the Simulink documentation.

Each type of lookup table block has its own set of options and associated tradeoffs. The examples in this section show how to use lookup tables effectively. The techniques demonstrated here will help you achieve maximal performance with minimal code and data sizes.

Multichannel Nonlinear Signal Conditioning

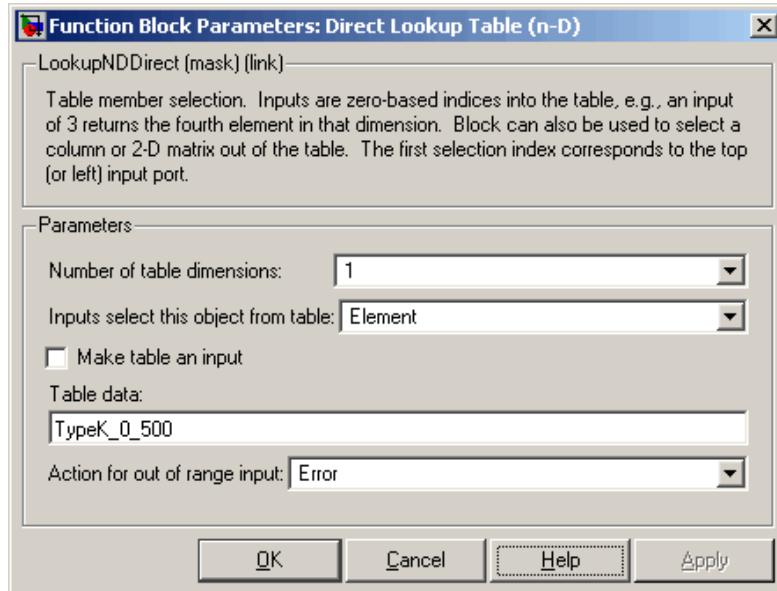
The next figure shows a Simulink model that reads input from two 8-channel, high-speed 8-bit analog-to-digital converters (ADCs). The ADCs are connected to Type K thermocouples through a gain circuit with an amplification of 250. Since the popular Type K thermocouples are highly nonlinear, there is an international standard for converting their voltages to temperature. In the range of 0 to 500 degrees Celsius, this conversion is a tenth-order polynomial. One way to perform the conversion from ADC readings (0-255) into temperature (in degrees Celsius) is to evaluate this polynomial. In the best case, the polynomial evaluation requires 9 multiplications and 10 additions per channel.

A polynomial evaluation is not the fastest way to convert these 8-bit ADC readings into measured temperature. Instead, the model uses a Direct Look-Up (n-D) Table block (named `TypeK_TC`) to map 8-bit values to temperature values. This block performs one array reference per channel.



Direct Look-Up Table (n-D) Block Conditions ADC Input

The block's table parameter has 256 values that correspond to the temperature at an ADC reading of 0, 1, 2, ... up to 255. The table data, calculated at the MATLAB prompt, is stored in the workspace variable `TypeK_0_500`. The block's **Table data** parameter field references `TypeK_0_500`, as the preceding figure shows.



Parameters of Direct Look-Up Table (n-D) Block

The model uses a Mux block to collect all similar signals (for example, Type K thermocouple readings) and feed them into a single Direct Look-Up Table block. This is more efficient than using one Direct Look-Up Table block per device. If multiple blocks share a common parameter (such as the table in this

example), the Real-Time Workshop code generator creates only one copy of that parameter in the generated code.

This is the recommended approach for signal conditioning when the size of the table can fit within your memory constraints. In this example, the table stores 256 double (8-byte) values, utilizing 2 KB of memory.

The TypeK_TC block processes 24 channels of data sequentially.

The Real-Time Workshop product generates the following code for the TypeK_TC block shown previously.

```
/* LookupNDDirect: '<Root>/TypeK_TC' */
/* 1-dimensional Direct Look-Up returning 24 Scalars */
{
    int_T i1;

    const uint8_T *u0 = rtb_TmpHiddenBuffer_Feeding_Typ;
    real_T *y0 = rtb_TypeK_TC_k;

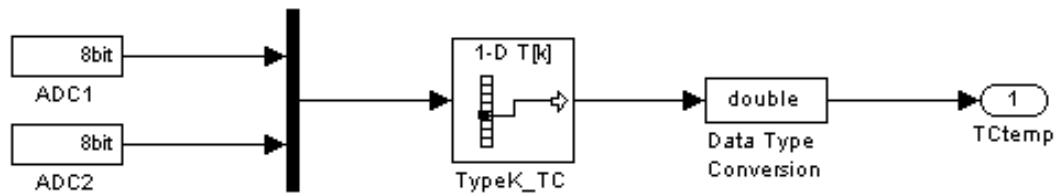
    for (i1=0; i1 < 24; i1++) {
        y0[i1] = (lookupADC_ConstP.TypeK_TC_table[u0[i1]]);
    }
}

{
    int32_T i1;

    /* Outport: '<Root>/TCtemp' */
    for(i1=0; i1<24; i1++) {
        lookupADC_Y.TCtemp[i1] = rtb_TypeK_TC_k[i1];
    }
}
```

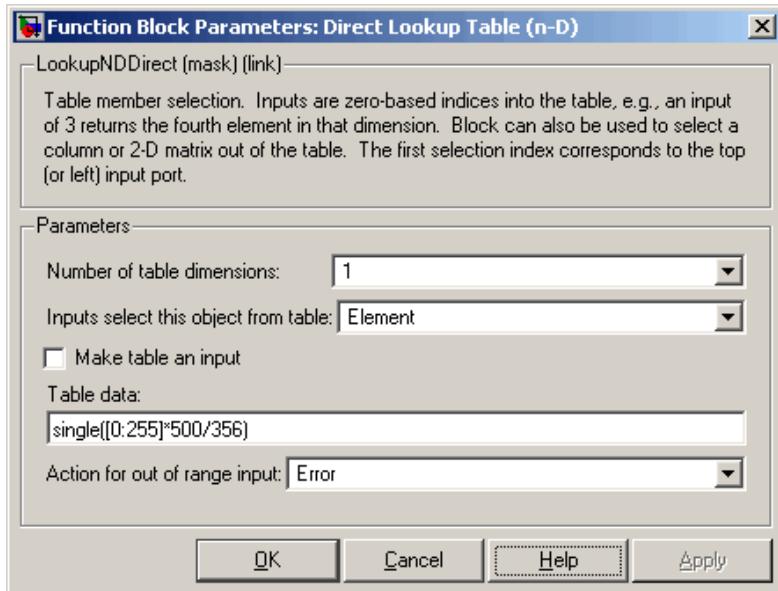
Notice that the core of each loop is one line of code that directly retrieves a table element from the table and places it in the block output variable. There are two loops in the generated code because the two simulated ADCs are not merged into a contiguous memory array in the Mux block. Instead, to avoid a copy operation, the Direct Look-Up Table block performs the lookup on two sets of data using a single table array (`lookupADC_ConstP.TypeK_TC_table[]`).

If the input accuracy for your application (not to be confused with the number of I/O bits) is 24 bits or less, you can use a single-precision table for signal conditioning. Then, cast the lookup table output to double precision for use in the rest of the block diagram. This technique, shown in the next section, causes no loss of precision.



Single-Precision Lookup Table Output Is Cast to Double Precision

A direct lookup table covering 24 bits of accuracy would require 64 megabytes of memory, which is typically not practical. To create a single-precision table, use the MATLAB `single()` cast function in your table calculations. Alternatively, you can perform the typecast directly in the **Table data** parameter, as shown in the next section.



Typecasting Table Data in a Direct Lookup Block

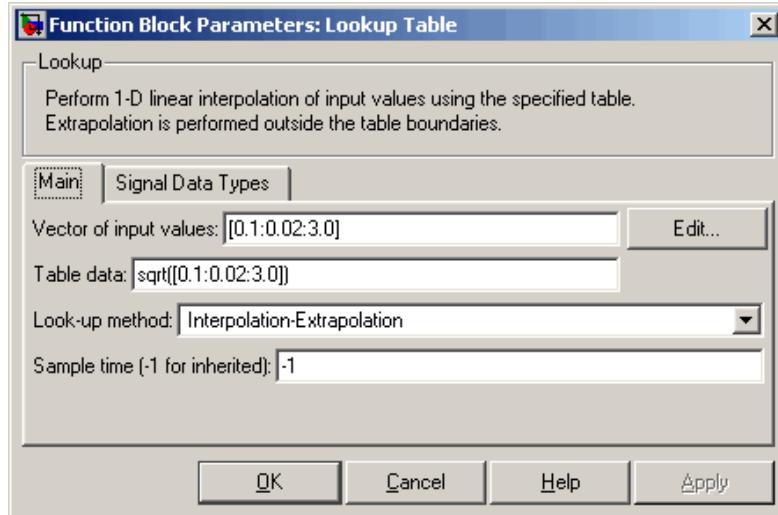
When table size becomes impractical, you must use other nonlinear techniques, such as interpolation or polynomial techniques. The Look-Up Table (n-D) block supports linear interpolation and cubic spline interpolation. The Polynomial block supports evaluation of noncomplex polynomials.

Compute-Intensive Equations

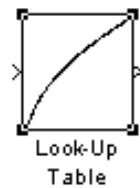
The blocks described in this section are useful for simplifying fixed, complex relationships that are normally too time consuming to compute in real time.

The only practical way to implement some compute-intensive functions or arbitrary nonlinear relationships in real time is to use some form of lookup table. On processors that do not have floating-point instructions, even functions like `sqrt()` can become too expensive to evaluate in real time.

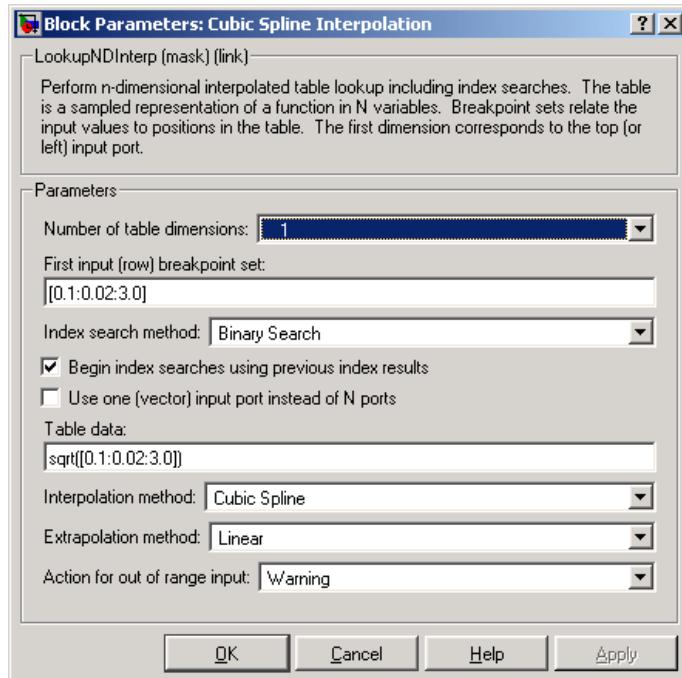
An approximation to the nonlinear relationship in a known range will work in most cases. For example, your application might require a square root calculation that your target processor's instruction set does not support. The next figure shows how you can use a Look-Up Table block to calculate an approximation of the square root function that covers a given range of the function.



The interpolated values are plotted on the block icon.



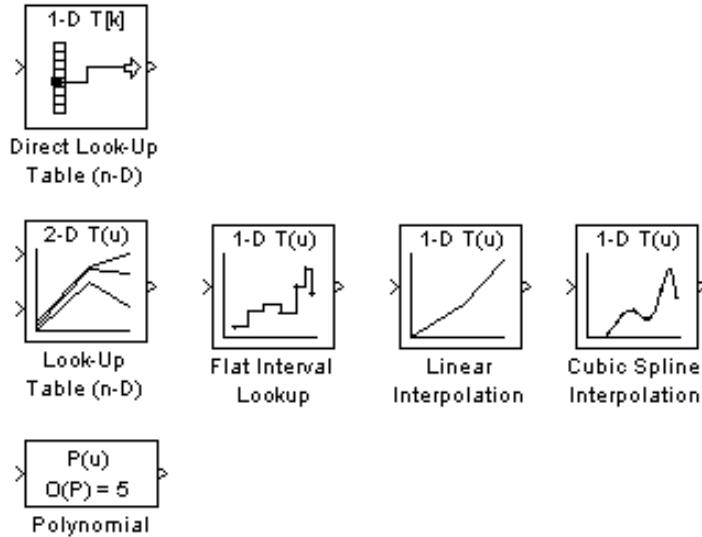
For more accuracy on widely spaced points, use a cubic spline interpolation in the Look-Up Table (n-D) block, as shown in the next figure.



Techniques available in the Simulink product include n-dimensional support for direct lookup, linear interpolations in a table, cubic spline interpolations in a table, and 1-D real polynomial evaluation.

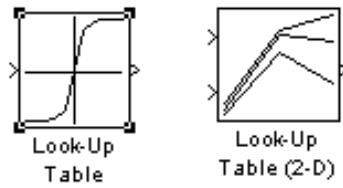
The Look-Up Table (n-D) block supports flat interval lookup, linear interpolation and cubic spline interpolation. Extrapolation for the Look-Up Table (n-D) block can either be disabled (clipping) or enabled for linear or spline extrapolations.

The icons for the Direct Look-Up Table (n-D) and Look-Up Table (n-D) blocks change depending on the type of interpolation selected and the number of dimensions in the table, as shown in the next figure.



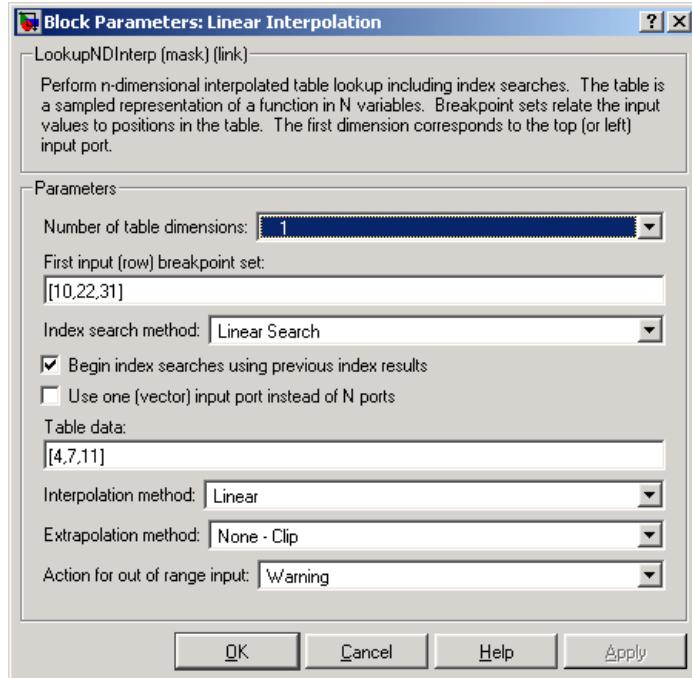
Tables with Repeated Points

The Look-Up Table and Look-Up Table (2-D) blocks, shown in the next figure, support linear interpolation with linear extrapolation. In these blocks, the row and column parameters can have repeated points, allowing pure step behavior to be mixed in with the linear interpolations. This capability is not supported by the Look-Up Table (n-D) block.



Slowly Versus Rapidly Changing Look-Up Table Block Inputs

You can optimize lookup table operations using the Look-Up Table (n-D) block for efficiency if you know the input signal's normal rate of change. The next figure shows the parameters for the Look-Up Table (n-D) block.

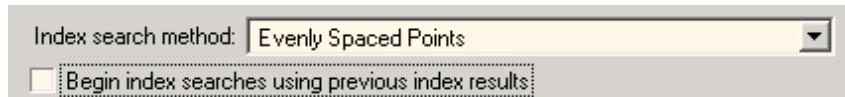


Parameter Dialog Box for the Look-Up Table (n-D) Block

If you do not know the input signal's normal rate of change in advance, it would be better to choose the **Binary Search** option for the index search in the Look-Up Table (n-D) block and the PreLook-Up Index Search block.



Regardless of signal behavior, if the table's breakpoints are evenly spaced, it is best to select the **Evenly Spaced Points** option from the Look-Up Table (n-D) block's parameter dialog box.



If the breakpoints are not evenly spaced, first decide which of the following best describes the input signal behavior.

- Behavior 1: The signal stays in a given breakpoint interval from one time step to the next. When the signal moves to a new interval, it tends to move to an adjacent interval.
- Behavior 2: The signal has many discontinuities. It jumps around in the table from one time step to the next, often moving three or more intervals per time step.

Given behavior 1, the best optimization for a given lookup table is to use the **Linear search option** and **Begin index searches using previous index results** options, as shown in the next figure.

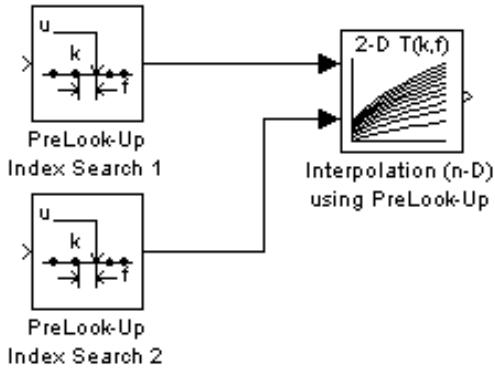


Given behavior 2, the **Begin index searches using previous index results** option does not necessarily improve performance. Choose the **Binary Search** option, as shown in the next figure.



The choice of an index search method can be more complicated for lookup table operations of two or more dimensions with linear interpolation. In this case, several signals are input to the table. Some inputs may have evenly spaced points, while others can exhibit behavior 1 or behavior 2.

Here it might be best to use PreLook-Up Index Search blocks with different search methods (evenly spaced, linear search, or binary search) chosen according to the input signal characteristics. The outputs of these search blocks are then connected to an Interpolation (n-D) Using PreLook-Up Index Search block, as shown in the block diagram in the next figure.

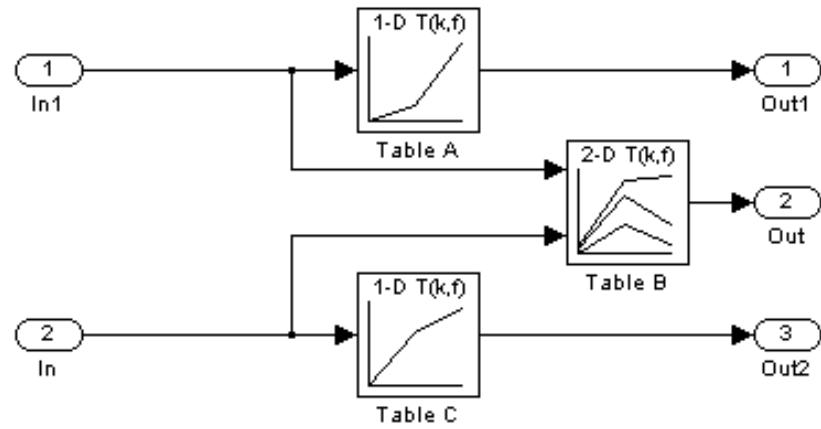


You can configure each PreLook-Up Index Search block independently to use the best search algorithm for the breakpoints and input time variation cases.

Multiple Tables with Common Inputs

The index search can be the most time consuming part of flat or linear interpolation calculations. In large block diagrams, lookup table blocks often have the same input values as other lookup table blocks. If this is the case in your block diagram, you can save much computation time by making the breakpoints common to all tables. This saving is obtained by using one set of PreLook-Up Index Search blocks to perform the searches once for all tables, so that only the interpolation remains to be calculated.

The next figure is an example of a block diagram that can be optimized by this method.

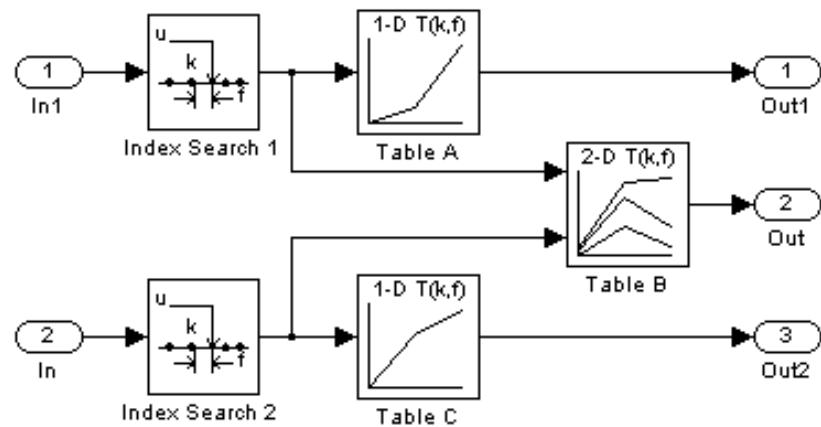


Before Optimization

Assume that the breakpoints for Table A are the same as the first input breakpoints for Table B, and that the breakpoints for Table C are the same as the second input breakpoints for Table B.

A 50% reduction in index search time is obtained by pulling these common breakpoints out into a pair of PreLook-Up Index Search blocks, and using Interpolation (n-D) Using PreLook-Up Index Search blocks to perform the interpolation.

The next figure shows the optimized block diagram.



After Optimization

In the above diagram, the Look-Up Table (n-D) blocks have been replaced with Interpolation (n-D) Using PreLook-Up blocks. The PreLook-Up Index Search blocks have been added to perform the index searches separately from the interpolations, to realize the savings in computation time.

In large controllers and simulations, it is not uncommon for hundreds of multidimensional tables to rely on a dozen or so breakpoint sets. Using the optimization technique shown in this example, you can greatly increase the efficiency of your application.

Reducing the Number of Blocks in a Model

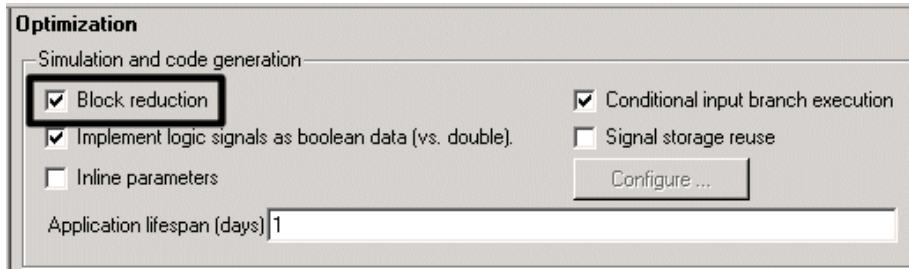
When you check **Block reduction**, the Simulink engine collapses certain groups of blocks into a single, more efficient block, or removes them entirely. This results in faster execution during model simulation and in generated code. The appearance of the source model does not change.

Note that block reduction is only intended to remove the code that represents execution of a block. Other supporting data, such as definitions for sample time and data types might remain in the generated code.

By default, **Block reduction** is checked. The types of block reduction are

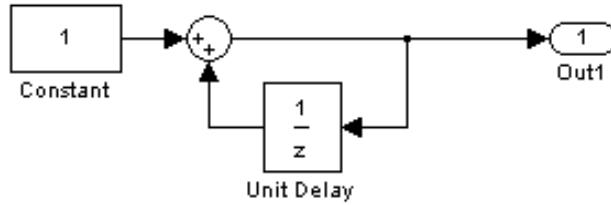
- “Replacing Accumulator Constructs” on page 26-29
- “Removing Redundant Type Conversions” on page 26-30
- “Eliminating Dead Code” on page 26-31
- “Removing Fast-to-Slow Rate Transition Blocks in Single-Tasking Systems” on page 26-32

To enable this option, select **Block reduction** on the **Optimization** pane of the Configuration Parameters dialog box, as shown in the next figure.



Replacing Accumulator Constructs

The Simulink engine recognizes the block diagram shown in the next figure as an accumulator. An accumulator construct — comprising a Constant block, a Sum block, and feedback through a Unit Delay block — is recognized anywhere across a block diagram, or within subsystems at lower levels.



An Accumulator Algorithm

With the **Block reduction** option enabled, the Simulink engine creates a synthesized block, `Sum_synth_accum`. The synthesized block replaces the previous block diagram, resulting in a simple increment calculation.

```
static void accum_output(int_T tid)
{
    /* UnitDelay Block: '<Root>/Unit Delay'
     *   Operating as an accumulator
     */
    accum_DWork.UnitDelay_DSTATE++;
    accum_B.UnitDelay_j = accum_DWork.UnitDelay_DSTATE;

    /* Outport: '<Root>/Out1' */
    accum_Y.Out1 = accum_B.UnitDelay_j;
}
```

With **Block reduction** turned off, the generated code reflects the block diagram more literally, but less efficiently.

```
static void accum_output(int_T tid)
{
    /* UnitDelay: '<Root>/Unit Delay' */
    accum_B.UnitDelay_j = accum_DWork.UnitDelay_DSTATE;

    /* Sum: '<Root>/Sum' */
    accum_B.Sum_1 = 1.0 + accum_B.UnitDelay_j;

    /* Outport: '<Root>/Out1' */
    accum_Y.Out1 = accum_B.Sum_1;
}
```

Removing Redundant Type Conversions

Unnecessary type conversion blocks are removed. For example, an `int` type conversion block whose input and output are of type `int` is redundant and is removed.

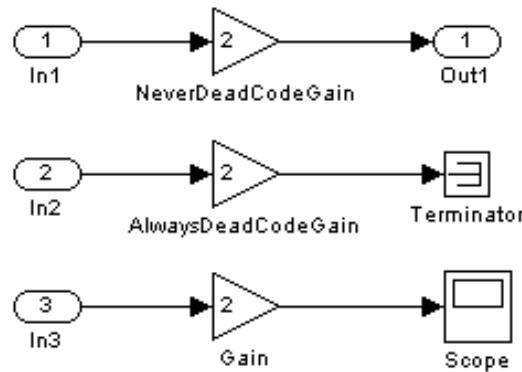
Eliminating Dead Code

Any blocks or signals in an *unused code path* are eliminated from generated code. The following conditions need to be met for a block to be considered part of an unused code path:

- All signal paths for the block end with a block that does not execute. Examples of blocks that do not execute include Terminator blocks, disabled Assertion blocks, S-Function blocks configured for block reduction, and To Workspace blocks when MAT-file logging is disabled for code generation.
- No signal paths for the block include global signal storage downstream from the block.

Tunable parameters do not prevent a block from being reduced by dead code elimination.

Consider the signal paths in the following block diagram.



If you check **Block reduction**, the Real-Time Workshop code generator responds to each signal path as follows:

For Signal Path...	The Real-Time Workshop Code Generator...
In1 to Out1	Always generates code because dead code elimination conditions are not met.
In2 to Terminator	Never generates code because dead code elimination conditions are met.
In3 to Scope	Generates code if MAT-file logging is enabled and eliminates code if MAT-file logging is disabled.

Removing Fast-to-Slow Rate Transition Blocks in Single-Tasking Systems

In a single-tasking system, if the **Block reduction** option is on, fast-to-slow Rate Transition blocks are optimized out of the model. For an example of single-tasking execution with fast-to-slow Rate Transition blocks optimized out, see “Single-Tasking Execution” on page 5-28.

Optimizing Code for Switch Blocks

You can optimize simulation and code generation for Switch and Multiport Switch blocks by enabling conditional input branch execution. By default, the Real-Time Workshop code generation parameters are configured to use the conditional input branch optimization.

When conditional input branch optimization is on, instead of executing all blocks driving the Switch block input ports at each time step, only the blocks required to compute the control input and the data input selected by the control input are executed.

Several considerations affect or limit Switch block optimization:

- Only blocks with -1 (inherited) or inf (Constant) sample time can participate in Switch block optimization.
- Blocks with outputs flagged as test points cannot participate.
- No multirate block can participate.
- Blocks with states cannot participate.

- Only S-functions with option `SS_OPTION_CAN_BE_CALLED_CONDITIONALLY` set can participate.

You control conditional input branch optimization by selecting and deselecting the **Conditional input branch execution** parameter on the **Optimization** pane of the Configuration Parameters dialog box.

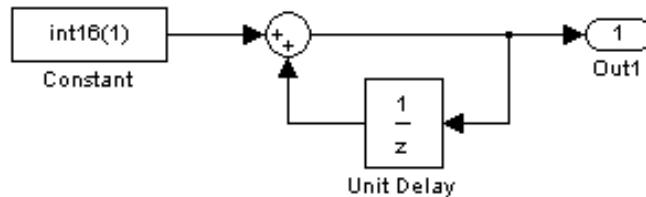
To run a conditional input branch optimization demo, click `rtwdemo_condininput` or type the following command at the MATLAB prompt.

```
rtwdemo_condininput
```

Optimizing Data Type Usage

In most processors, the use of integer data types can result in a significant reduction in data storage requirements, as well as a large increase in the speed of operation. You can achieve large performance gains on most processors by identifying those portions of your block diagram that are really integer calculations (such as accumulators), and implementing them with integer data types. Floating-point DSP targets are an obvious exception to this rule.

The accumulator from the previous example used 64-bit floating-point calculations by default. The block diagram in the next figure implements the accumulator with 16-bit integer operations.



Accumulator Implemented with 16-bit Integers

If the **Saturate on integer overflow** option of the Sum block is turned off, the code generated from the integer implementation looks the same as code generated from the floating-point block diagram. However, since `Sum_synth_accum` is performing integer arithmetic internally, the accumulator executes more efficiently.

By default, the **Saturate on integer overflow** option is on. This option generates extra error-checking code from the integer implementation, as shown in the following example.

```
static void accum_int16_output(int_T tid)
{
    /* local block i/o variables */

    int16_T rtb_UnitDelay_k;

    /* UnitDelay: '<Root>/Unit Delay' incorporates:
     *   Constant: '<Root>/Constant'
     *
     * Regarding '<Root>/Unit Delay':
     *   Operating as an accumulator
     */
    {
        int16_T tmpVar = accum_int16_DWork.UnitDelay_DSTATE;
        accum_int16_DWork.UnitDelay_DSTATE = tmpVar + (1);
        if ((tmpVar >= 0) && ((1) >= 0) &&
            (accum_int16_DWork.UnitDelay_DSTATE < 0))
        {
            accum_int16_DWork.UnitDelay_DSTATE = MAX_int16_T;
        } else if ((tmpVar < 0) && ((1) < 0) &&
                   (accum_int16_DWork.UnitDelay_DSTATE >= 0)) {
            accum_int16_DWork.UnitDelay_DSTATE = MIN_int16_T;
        }
    }
    rtb_UnitDelay_k = accum_int16_DWork.UnitDelay_DSTATE;

    /* Outport: '<Root>/Out1' */
    accum_int16_Y.Out1 = rtb_UnitDelay_k;
}
```

The floating-point implementation would not have generated the saturation error checks, which apply only to integers. When using integer data types, consider whether or not you need to generate saturation checking code.

If you are able to ignore saturation checks, turn **Saturate on integer overflow** off for the Sum block. The generated code then omits the preceding checks:

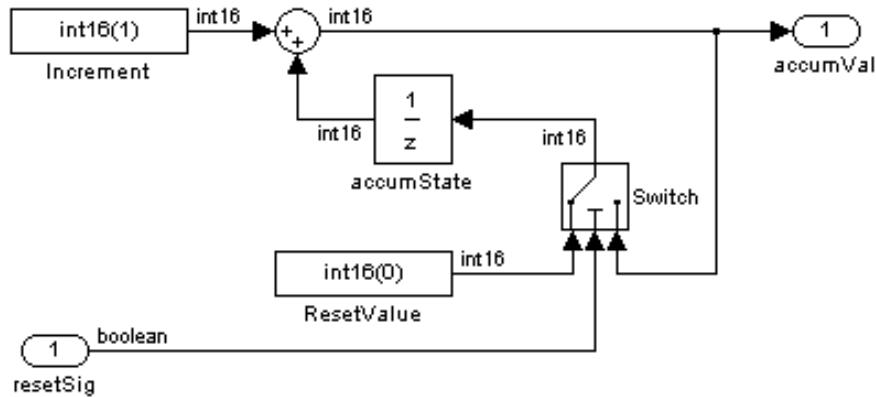
```
static void accum_int16_output(int_T tid)
{
    /* local block i/o variables */

    int16_T rtb_UnitDelay_k;

    /* UnitDelay: '<Root>/Unit Delay' incorporates:
     *   Constant: '<Root>/Constant'
     *
     * Regarding '<Root>/Unit Delay':
     *   Operating as an accumulator
     */
    accum_int16_DWork.UnitDelay_DSTATE++;
    rtb_UnitDelay_k = accum_int16_DWork.UnitDelay_DSTATE;

    /* Outport: '<Root>/Out1' */
    accum_int16_Y.Out1 = rtb_UnitDelay_k;
}
```

The next figure shows an efficient way to add a reset to an integer accumulator. When `resetSig` is greater than or equal to the threshold of the Switch block, the Switch block passes the reset value (0) back into the accumulator.



The reset signal can protect computations from overflows, and the size of the resultant code is minimal. The code uses no floating-point operations.

```

static void accum_rst_output(int_T tid)
{
    /* local block i/o variables */

    int16_T rtb_Sum_j;

    /* Sum: '<Root>/Sum' */
    rtb_Sum_j = (int16_T)(1 + accum_rst_DWork.accumState_DSTATE);

    /* Outport: '<Root>/accumVal' */
    accum_rst_Y.accumVal = rtb_Sum_j;

    /* Switch: '<Root>/Switch' */
    if(accum_rst_U.resetSig) {
        accum_rst_B.Switch_k = 0;
    } else {
        accum_rst_B.Switch_k = rtb_Sum_j;
    }
}

```

This example uses an input to the system as the reset value, but you can also use an `int16` constant.

Note You should not use preprocessor compile-time mechanisms to redefine the Real-Time Workshop data types used in generated code. Redefining data type size or sign properties, using such mechanisms, can affect numerical results or cause runtime exceptions due to the data not matching properties expected by the generated code. Instead, use the **Hardware Implementation** pane of the Configuration Parameters dialog box to specify the appropriate word size for the model and specify the desired data types (`double`, `single`, `int32`, ...) for signals and parameters within the model.

Additional Integer and Fixed-Point Optimizations

You may find several companion products useful in optimizing the performance and size of integer-based generated code.

Generating Pure Integer Code with the Real-Time Workshop Embedded Coder Target

The Real-Time Workshop Embedded Coder target (ERT) provides the **Support floating-point numbers** option (the inverse of an obsolete option named **Integer code only**) to control whether generated code contains any floating-point data or operations. When this option is deselected, an error is raised if any noninteger data or expressions are encountered during compilation of the model. The error message reports the offending blocks and parameters.

If pure integer code generation is important to your design, you should consider using the Real-Time Workshop Embedded Coder target (or a target of your own, based on the Real-Time Workshop Embedded Coder target).

The Real-Time Workshop Embedded Coder target offers many additional optimizations. See the Real-Time Workshop Embedded Coder documentation for more information.

Optimizing Integer Code with the Simulink Fixed Point and Stateflow Products

The Simulink Fixed Point product is designed to deliver the highest levels of performance for noninteger algorithms on fixed-point processors. If you have a Simulink Fixed Point license, you can use Simulink products to simulate

effects commonly encountered in fixed-point systems for applications such as control systems and time-domain filtering. In addition, you can generate C code for execution on a fixed-point embedded processor with the Real-Time Workshop code generator. The generated code uses only integer types and automatically includes all operations, such as shifts, needed to account for differences in fixed-point locations. The code generation strategy maps the integer value set to a range of expected real-world values to achieve high efficiency.

Finite-state machine or flowchart constructs can often represent decision logic (or mode logic) efficiently. Stateflow charts, which are fully integrated into Simulink models, provide these capabilities and support integer data-typed code generation.

Optimizing Signals

In this section...

- “Implementing Logic Signals as Boolean Data” on page 26-39
- “Reducing Memory Requirements for Signals” on page 26-39
- “Declaring Signals as Local Function Data” on page 26-40
- “Reusing Memory Allocated for Signals” on page 26-40
- “Inlining Invariant Signals” on page 26-41

Implementing 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)”.

Reducing Memory Requirements for Signals

To instruct the Real-Time Workshop code generator 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 Chapter 11, “Signal Considerations”.

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 in the **Optimization** pane. See “Declaring Signals as Local Function Data” on page 26-40 and “Reusing Memory Allocated for Signals” on page 26-40.

Declaring 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 Chapter 11, “Signal Considerations”. Also see “First Look at Generated Code” in Getting Started.

Reusing 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 Real-Time Workshop code generator reuses signal memory whenever possible. When **Reuse block output** is cleared, signals are stored in unique locations.

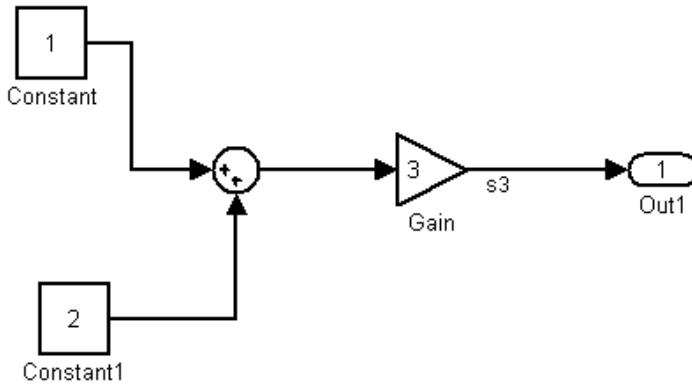
Reuse block output is enabled only when you select **Signal storage reuse**.

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 Chapter 11, “Signal Considerations” for more information (including generated code example) on **Reuse block output**, **Minimize data copies between local and global variables**, and other signal storage options.

Inlining 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** pane, the Real-Time Workshop code generator inlines the invariant signal **S3** in the generated code.

Note that an *invariant signal* is not the same as an *invariant constant*. (See “Optimization Pane” in the Simulink documentation for information on invariant constants.) 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, **Inline parameters** must be *on* for all referenced models. If a referenced model does not have **Inline Parameters** set to *on*, the Simulink engine temporarily enables this option while generating code for the referenced model, then turns it off again when the build completes. Thus the referenced model is left in its previous state and need not be resaved. For the , **Inline parameters** can be either *on* or *off*.

Inlining Parameters

When you select the **Inline parameters** configuration parameter:

- The Real-Time Workshop 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. An inlined parameter, because it has in effect been transformed into a constant, is no longer tunable. That is, it is not visible to externally written code, and its value cannot be changed at run-time.

- 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 Real-Time Workshop 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 Chapter 10, “Parameter Considerations” 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 performance 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 *on*. See “Inlining Invariant Signals” on page 26-41.

Referenced Models

When a top model uses referenced models,

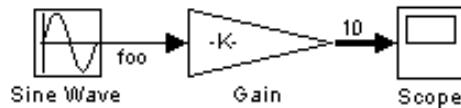
- All referenced models must specify **Inline parameters** to be *on*
- The top model can specify **Inline parameters** to be *on* or *off*.

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. See "Using Model Arguments" in the Simulink documentation for details.

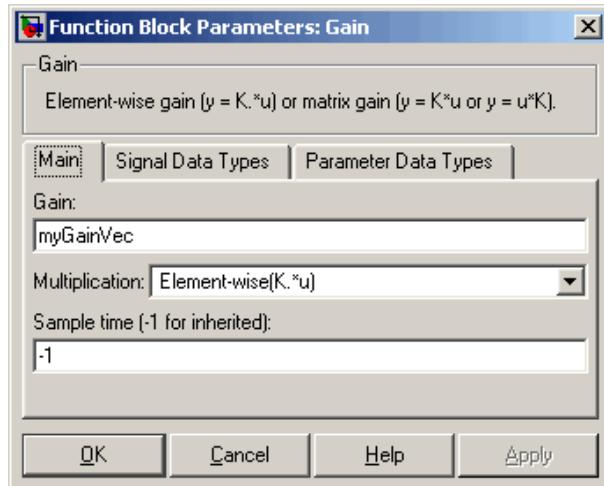
Configuring a Loop Unrolling Threshold

The **Loop unrolling threshold** parameter on the **Optimization** 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 the Target Language Compiler documentation for more information on loop rolling.

Note When a model includes Stateflow charts or Embedded MATLAB Function blocks, a set of Stateflow optimizations appears on the **Optimization** pane. The settings you make for the Stateflow options also apply to all Embedded MATLAB Function blocks in the model. This is because the Embedded 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 Embedded MATLAB Function blocks.

Optimizing Code Generated for Vector Assignments

In this section...

[“Overview” on page 26-47](#)

[“Example: Using memcpy for Vector Assignments” on page 26-48](#)

Overview

The **Use memcpy for vector assignment** option lets you optimize Real-Time Workshop 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 use this model option to specify that generated code should 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 the *Simulink Graphical User Interface* documentation.

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 any 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.

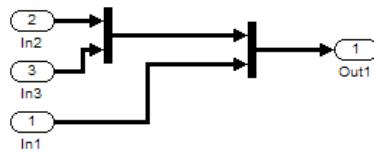
Note The `memcpy` optimization may not occur under certain conditions, including when other optimizations have a higher precedence than the `memcpy` optimization, or when the generated code is originating from Target Language Compiler (TLC) code, such as a TLC file associated with an S-function block.

Note If you are licensed for Real-Time Workshop Embedded Coder software, you can use a target function library (TFL) to provide your own custom implementation of the `memcpy` function to be used in generated model code. For more information, see “Example: Mapping the `memcpy` Function to a Target-Specific Implementation” in the Real-Time Workshop Embedded Coder documentation.

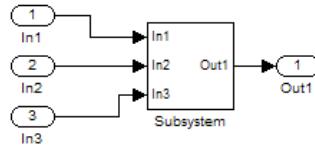
Example: Using `memcpy` for Vector Assignments

To examine the effect on generated vector assignment code of using the **Use `memcpy` for vector assignment** option, 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** 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.mdl`.
- 2** Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the **Create code generation report**. Then go to the **Real-Time Workshop** 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 `model.c` section (for example, `vectorassign.c`) and inspect the model output function. Notice that the vector assignments are implemented using for loops.

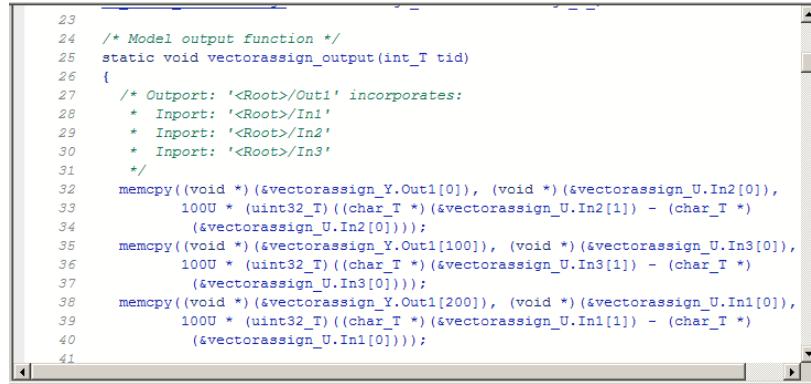
```

24  /* Model output function */
25  static void vectorassign_output(int_T tid)
26  {
27      {
28          int32_T i;
29
30          /* Outport: '<Root>/Out1' incorporates:
31          *   Import: '<Root>/In1'
32          *   Import: '<Root>/In2'
33          *   Import: '<Root>/In3'
34          */
35          for (i = 0; i < 100; i++) {
36              vectorassign_Y.Out1[i] = vectorassign_U.In2[i];
37          }
38
39          for (i = 0; i < 100; i++) {
40              vectorassign_Y.Out1[i + 100] = vectorassign_U.In3[i];
41          }
42
43          for (i = 0; i < 100; i++) {
44              vectorassign_Y.Out1[i + 200] = vectorassign_U.In1[i];
45          }
46      }
  
```

- 4** Go to the **Optimization** 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 *model.c* section and inspect the model output function. Notice that the vector assignments now are implemented using `memcpy` function calls.



```
23
24 /* Model output function */
25 static void vectorassign_output(int_T tid)
26 {
27     /* Outport: '<Root>/Out1' incorporates:
28      * Inport: '<Root>/In1'
29      * Inport: '<Root>/In2'
30      * Inport: '<Root>/In3'
31      */
32     memcpy((void *)(&vectorassign_Y.Out1[0]), (void *)(&vectorassign_U.In2[0]),
33            100U * (uint32_T)((char_T *)(&vectorassign_U.In2[1]) - (char_T *)
34            (&vectorassign_U.In2[0])));
35     memcpy((void *)(&vectorassign_Y.Out1[100]), (void *)(&vectorassign_U.In3[0]),
36            100U * (uint32_T)((char_T *)(&vectorassign_U.In3[1]) - (char_T *)
37            (&vectorassign_U.In3[0])));
38     memcpy((void *)(&vectorassign_Y.Out1[200]), (void *)(&vectorassign_U.In1[0]),
39            100U * (uint32_T)((char_T *)(&vectorassign_U.In1[1]) - (char_T *)
40            (&vectorassign_U.In1[0])));
41 }
```

Controlling 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 5-80.

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 a lifespan no longer than necessary, and a step size no smaller than necessary.

An application is guaranteed to be able to run to its specified lifespan. It may be able to run longer, but is not guaranteed to do so. 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 Real-Time Workshop length limitation of 32 bits because the value of a time counter never provides the value of a signal, state, or parameter.

For information about the allocation and operation of absolute and elapsed time counters, see “Using Timers” on page 5-70. For information about asynchronous timing, see “Using Timers in Asynchronous Tasks” on page 5-57. For information about the effect of the **Application lifespan (days)** parameter on simulation, see “Application lifespan (days)” in the Simulink documentation.

Using 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 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 Real-Time Workshop 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 the Target Language Compiler document.

Optimizing Code Resulting from Floating-Point to Integer Conversions

In this section...

[“Removing Code That Wraps Out-of-Range Values” on page 26-54](#)

[“Removing Code That Maps NaN Values to Integer Zero” on page 26-55](#)

Removing 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 Real-Time Workshop code generator to remove code that ensures that execution of the generated code 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](#)” in the *Simulink Graphical User Interface*.

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), 4.294967296000000E+009);
if(_fixptlowering1 < -2.147483648000000E+009) {
    _fixptlowering1 += 4.294967296000000E+009;
} else if(_fixptlowering1 >= 2.147483648000000E+009) {
    _fixptlowering1 -= 4.294967296000000E+009;
}
cg_in_0_20_0[i1] = (int32_T)_fixptlowering1;
```

The code generator applies the `fmod` function to handle out-of-range conversion results.

Removing Code That Maps NaN Values 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 Real-Time Workshop code generator to remove code that ensures that execution of the generated code 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” in the *Simulink Graphical User Interface*.

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;
}
```

Improving Efficiency of Code That Uses Division by Constant Power of 2

In this section...

“How to Improve Efficiency of Code That Uses Division by Constant Power of 2” on page 26-56

“Rationale for Using Division or Shifts in Generated Code” on page 26-57

“When Does Real-Time Workshop Replace Division by Constant Power of 2 with Shifts?” on page 26-57

“When Does Real-Time Workshop Not Replace Division by Constant Power of 2 with Shifts?” on page 26-58

How to Improve Efficiency of Code That Uses Division by Constant Power of 2

Generally, the simplest rounding mode produces the most efficient generated code for division operations. However, this is not true for signed division where the denominator is an invariant constant and the value of the constant is a power of 2. In this case, you might see a division in your generated code instead of a shift. (To learn why, see “Rationale for Using Division or Shifts in Generated Code” on page 26-57.) If a shift is more suitable for your target hardware, selecting rounding to floor replaces the division with a shift provided that the following conditions apply:

- Division by an invariant constant power of 2.
- Fully inlined constants. If you do not inline parameters, the denominator appears as a variable in the generated code and is not optimized. See “Inline parameters” in the *Simulink Graphical User Interface*.
- Hardware Signed integer division rounds to configuration parameter is set to the correct behavior of your target hardware: either Zero or Floor. If not, simplest rounding on the block might result in undesired code, and code generation might not agree with simulation. See “Signed integer division rounds to” in the *Simulink Graphical User Interface*.
- If the constant is in a vector, all the elements of the vector must be exact powers of 2 and the size of the vector must be smaller than the Real-Time

Workshop loop unrolling threshold. See “Loop unrolling threshold” in the *Simulink Graphical User Interface*.

Rationale for Using Division or Shifts in Generated Code

When Real-Time Workshop generates code for a division by a constant power of 2, when possible, it attempts to replace the division with a shift. Real-Time Workshop does not automatically replace the division with a shift because it attempts to:

- Reduce the size of the generated code by avoiding rounding code and division helper functions.
- Allow for better target-dependent rounding code.
- Provide better global optimization opportunities for the target compiler.

When Does Real-Time Workshop Replace Division by Constant Power of 2 with Shifts?

In general, Real-Time Workshop replaces division with shifts if they are plain shifts without any additional rounding code. This is true for:

- Unsigned division

For unsigned operations, simplest rounding, rounding to floor, and rounding to zero are all equivalent. Therefore, Real-Time Workshop can always replace the division with a shift.

- Signed division with effective rounding to floor

For signed division with rounding to floor, or signed division with simplest rounding where the hardware `Signed integer division rounds to` configuration parameter is `Floor`, the effective rounding mode for the division is floor. Because a shift also rounds to floor, no additional rounding code is needed. Therefore, Real-Time Workshop replaces the division with a shift.

When Does Real-Time Workshop Not Replace Division by Constant Power of 2 with Shifts?

In general, Real-Time Workshop does not replace division with shifts when the shift is cumbersome. For example, for signed division with rounding to zero, or signed division with simplest rounding where the hardware `Signed integer division rounds to configuration parameter is Zero`, the effective rounding mode for the division is zero. On most systems, the simplest rounding method for division is round to zero. For signed operations, rounding to zero is not equivalent to rounding to floor. Because a shift rounds to floor, it needs additional rounding code to round to zero. Therefore, Real-Time Workshop generates a division, allowing the target compiler to perform the appropriate optimization. See “Example: Signed Division, Simplest Rounding” on page 26-59 and “Example: Signed Division, Rounding to Zero” on page 26-60.

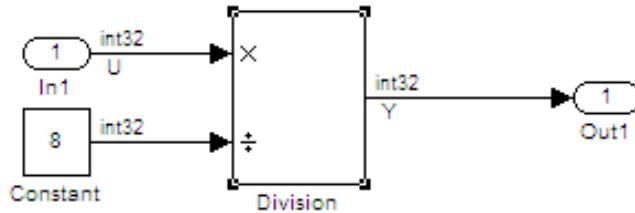
Exceptions

There are some exceptions for mixed signedness of division inputs:

- When the numerator is unsigned and the denominator is signed, Real-Time Workshop can use an unsigned data type to represent the denominator because it is a positive constant. Therefore, this case is equivalent to an unsigned division. Real-Time Workshop replaces the division with a shift.
- If the denominator is unsigned and its value is not representable using the data type of the numerator, Real-Time Workshop uses a shift with additional rounding code rather than a division operation with a helper function. See “Example: Division — Signed Numerator, Unsigned Denominator Does Not Fit in Signed” on page 26-61.

Note If the denominator is unsigned and its value is representable using the data type of the numerator, Real-Time Workshop can represent the denominator as a signed integer. Therefore, this case is equivalent to signed division. Real-Time Workshop generates a division to avoid a shift with additional rounding code.

Example: Signed Division, Simplest Rounding



This example demonstrates that changing the block rounding mode from simplest to floor results in more efficient generated code for this simple model.

To set up the model and generate code:

- 1** For the Import block **In1**, set the data type to **int32**
- 2** For the Constant block, set:
 - The output data type to **int32**.
 - The constant value to **8**.
- 3** For the Division block, set the integer rounding mode to **Simplest**.
- 4** Set the **Hardware Implementation > Embedded Hardware > Signed integer division rounds to** configuration parameter to **Zero**.
- 5** Select the **Real-Time Workshop > Report > Create code generation report** configuration parameter.
- 6** Select the **Optimization > Inline parameters** configuration parameter.
- 7** Generate code for the model. From the model main menu, select **Tools > Real-Time Workshop > Build Model**.
Real-Time Workshop generates code for the model and opens the Code Generation Report.
- 8** In the **Generated Files** list, click the link to open the C code file.

The generated code for the division is:

$$Y = U / 8;$$

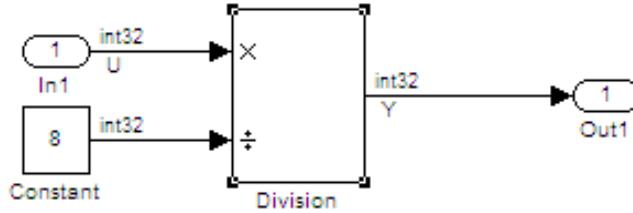
9 For the Division block, change the integer rounding mode to **Floor**.

10 Save the model and generate code.

The generated code for the division is now:

$$Y = U >> 3;$$

Example: Signed Division, Rounding to Zero



This example demonstrates that if the rounding mode is zero, the generated code contains a division.

To set up the model and generate code:

1 For the Input block **In1**, set the data type to **int32**.

2 For the Constant block, set:

- The output data type to **int32**.
- The constant value to **8**.

3 For the Division block, set the integer rounding mode to **Zero**.

4 Set the **Hardware Implementation > Embedded Hardware > Signed integer division rounds to** configuration parameter to **Zero**.

5 Select the **Real-Time Workshop > Report > Create code generation report** configuration parameter.

6 Select the **Optimization > Inline parameters** configuration parameter.

7 Generate code for the model. From the model main menu, select **Tools > Real-Time Workshop > Build Model**.

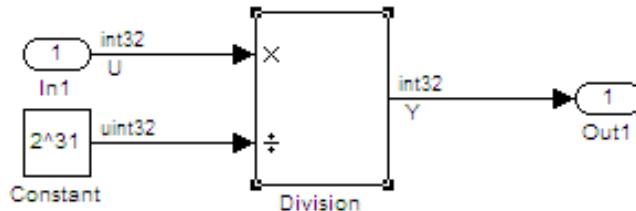
Real-Time Workshop generates code for the model and opens the Code Generation Report.

8 In the **Generated Files** list, click the link to open the C code file.

The generated code for the division is:

$$Y = U / 8;$$

Example: Division – Signed Numerator, Unsigned Denominator Does Not Fit in Signed



1 For the Input block **In1**, set the data type to **int32**.

2 For the Constant block, set:

- The output data type to **uint32**.
- The constant value to **2^31**.

3 For the Division block, set the integer rounding mode to **Simplest**.

4 Set the **Hardware Implementation > Embedded Hardware > Signed integer division rounds to** configuration parameter to **Zero**.

5 Select the **Real-Time Workshop > Report > Create code generation report** configuration parameter.

6 Select the **Optimization > Inline parameters** configuration parameter.

7 Generate code for the model. From the model main menu, select **Tools > Real-Time Workshop > Build Model**.

Real-Time Workshop generates code for the model and opens the Code Generation Report.

8 In the **Generated Files** list, click the link to open the C code file.

The generated code for the division is:

```
Y = ((U < 0 ? MAX_int32_T : 0) + U) >> 31;
```

Note Rounding to zero results in the same generated code.

9 For the Division block, change the integer rounding mode to **Floor**.

10 Save the model and generate code.

The generated code for the division is now:

```
Y = U >> 31;
```

Optimization Dependencies

Several parameters available on the **Optimization** pane have dependencies on settings of other options. The following table summarizes the dependencies.

Option	Dependencies?	Dependency Details
Block reduction	No	
Conditional input branch execution	No	
Implement logic signals as Boolean data (versus double)	Yes	Disable for models created with a Simulink version that supports only signals of type double
Signal storage reuse	No	
Inline parameters	Yes	Disable for referenced models in a model reference hierarchy
Application lifespan (days)	No	
Parameter structure (ERT targets only)	Yes	Enabled by Inline parameters
Enable local block outputs	Yes	Enabled by Signal storage reuse
Reuse block outputs	Yes	Enabled by Signal storage reuse
Inline invariant signals	Yes	Enabled by Inline parameters
Eliminate superfluous local variables (Expression folding)	Yes	Enabled by Signal storage reuse
Pack Boolean data into bitfields (ERT targets only)	No	
Bitfield declarator type specifier (ERT targets only)	Yes	Enabled by Pack Boolean data into bitfields
Minimize data copies between local and global variables	Yes	Enabled by Signal storage reuse
Simplify array indexing (ERT targets only)	No	
Loop unrolling threshold	No	

Option	Dependencies?	Dependency Details
Maximum stack size (bytes)	No	
Use memcpy for vector assignment	No	
Memcpy threshold (bytes)	Yes	Enabled by Use memcpy for vector assignment
Pass reusable subsystem output as (ERT targets only)	No	
Remove root level I/O zero initialization (ERT targets only)	No	
Use memset to initialize floats and doubles to 0.0	No	
Remove internal data zero initialization (ERT targets only)	No	
Optimize initialization code for model reference (ERT targets only)	Yes	Disable if model includes an enabled subsystem <i>and</i> the model is referred to from another model with a Model block
Remove code from floating-point to integer conversions that wrap out-of-range values	No	
Remove code from floating-point to integer conversions with saturation that maps NaN to zero	Yes (ERT targets) No (GRT targets)	For ERT targets, enabled by Support floating-point numbers and Support non-finite numbers in the Real-Time Workshop > Interface pane
Remove code that protects against division arithmetic exceptions (ERT targets only)	No	

Customizing Code Generation and the Build Process

- “Controlling the Compiling and Linking Phases of the Build Process” on page 27-2
- “Cross-Compiling Code Generated on a Microsoft Windows System” on page 27-4
- “Controlling the Location and Naming of Libraries During the Build Process” on page 27-7
- “Recompiling Precompiled Libraries” on page 27-13
- “Customizing Post Code Generation Build Processing” on page 27-14
- “Configuring Generated Code with TLC” on page 27-19
- “Customizing the Target Build Process with the STF_make_rtw Hook File” on page 27-22
- “Customizing the Target Build Process with sl_customization.m” on page 27-29
- “Replacing the STF_rtw_info_hook Mechanism” on page 27-34

Controlling the Compiling and Linking Phases of the Build Process

After generating code for a model, the Real-Time Workshop 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 Real-Time Workshop software generates code for the model, including a makefile.

- **Generate makefile** option

When you clear this option, the Real-Time Workshop software does not generate a makefile for the model. You must specify any post code generation processing, including compilation and linking, as a user-defined command, as explained in “Customizing Post Code Generation Build Processing” on page 27-14.

- **Makefile-only target**

The Microsoft Visual C++ Project Makefile versions of the `grt`, `grt_malloc`, and Real-Time Workshop 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 Real-Time Workshop software expands the |>TGT_FCN_LIB<| token to match the setting of the **Target function library** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box. Possible values for this option include ANSI_C, C99 (ISO), GNU99 (GNU), and C++ (ISO). You can use this token in a makefile conditional statement to specify compiler options to be used. For example, if you set the token to C99 (ISO), the compiler might need an additional option set to support C99 library functions.

Cross-Compiling Code Generated on a Microsoft Windows System

If you need to generate code with the Real-Time Workshop 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 directory. This will be your working directory 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 **Real-Time Workshop** 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 directory (directory from which the model was built) is not already Linux accessible, copy it to a Linux accessible path. For example, if your build directory for the generated code was D:\work\mymodel_grt_rtw, copy that entire directory tree to a path such as /home/user/mymodel_grt_rtw.
- 6 If the MATLAB directory tree on the Windows system is Linux accessible, skip this step. Otherwise, you must copy all the include and source directories to a Linux accessible drive partition, for example, /home/user/myinstall. These directories appear in the makefile after MATLAB_INCLUDES = and ADD_INCLUDES = and can be found by searching for \$(MATLAB_ROOT). Any path that contains \$(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 directory 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 directory that includes the subdirectories listed in the makefile. For example, the Signal Processing Blockset product requires the following directories to be copied:

```
$(MATLAB_ROOT)/toolbox/dspblk  
$(MATLAB_ROOT)/toolbox/rtw/dspblk
```

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 appropriate computer value, 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 directory (designating the location of the generated code on the Windows system) and parent directories to Linux accessible include directories. 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`).

Controlling the Location and Naming of Libraries During the Build Process

Two configuration parameters, `TargetPreCompLibLocation` and `TargetLibSuffix`, are available for you to use to control values placed in Real-Time Workshop 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 Real-Time Workshop library. Typically, a target has cross-compiled versions of these libraries and places them in a target-specific directory.
- 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 directory. For more information, see “Controlling the Location of Model Reference Libraries” on page 27-9.

Specifying the Location of Precompiled Libraries

Use the `TargetPreCompLibLocation` configuration parameter to:

- Override the precompiled library location specified in the `rtwmakecfg.m` file (see “Using the `rtwmakecfg.m` API to Customize Generated Makefiles” on page 32-93 for details)
- Precompile and distribute target-specific versions of product libraries (for example, the Signal Processing Blockset product)

For a precompiled library, such as a blockset library or the Real-Time Workshop library, the location specified in `rtwmakecfg.m` is typically a location specific to the blockset or the Real-Time Workshop product. It is expected that the library will exist in this location and it is linked against during Real-Time Workshop 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 Real-Time Workshop build process. All 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 Real-Time Workshop 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 Real-Time Workshop build process uses the location specified in `rtwmakecfg.m` for the token replacement.

Controlling the Location of Model Reference Libraries

On platforms other than the Apple® Macintosh® platform, when building a model that uses referenced models, the Real-Time Workshop build process by default:

- Copies libraries used by the referenced models to the parent model's build directory
- Assigns the filenames of the libraries to `MODELREF_LINK_LIBS` in the generated makefile

For example, if a model includes a referenced model `sub`, the Real-Time Workshop build process assigns the library name `sub_rtwlib.lib` to `MODELREF_LINK_LIBS` and copies the library file to the parent model's build directory. 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 Real-Time Workshop build process:

- Does not copy libraries used by the referenced models to the parent model's build directory
- Assigns the relative paths and filenames of the libraries to **MODELREF_LINK_LIBS** in the generated makefile

When using this technique, the Real-Time Workshop 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 **Real-Time Workshop** pane of the Configuration Parameters dialog box:

```
make_rtw USE_MDLREF_LIBPATHS=1
```

If you specify other Make command arguments, such as `OPTS="-g"`, you can specify the multiple arguments in any order.

To return to the default behavior, set `USE_MDLREF_LIBPATHS` to 0, or remove it.

Controlling 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 affect the suffix of both precompiled and non-precompiled libraries configured from the `rtwmakecfg` API. For details, see “Using the `rtwmakecfg.m` API to Customize Generated Makefiles” on page 32-93.

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 Real-Time Workshop 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 Real-Time Workshop 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 Real-Time Workshop build process includes `TargetLibSuffix` implicitly. For example, for a top model named `topmodel` with submodels named `submodel1` and `submodel2`, 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 = \
submodel1_rtwlib_diab.library submodel2_rtwlib_diab.library
```

By default, the `TargetLibSuffix` parameter is an empty string. In this case, the Real-Time Workshop 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:

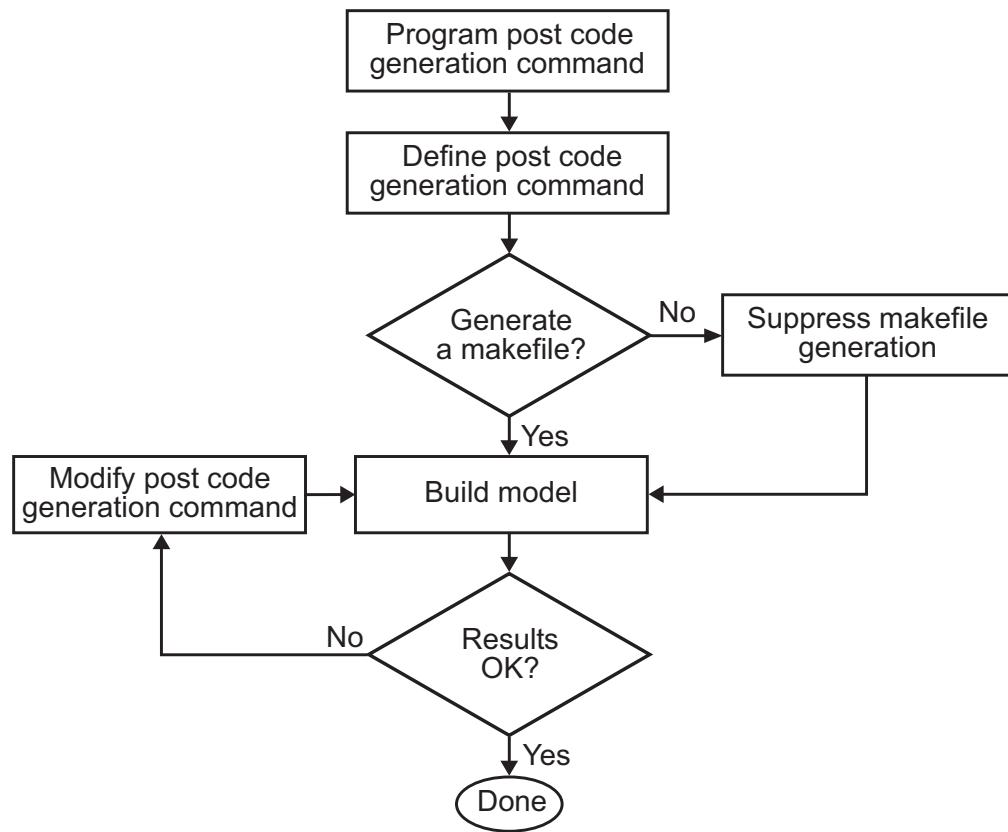
```
SHARED_LIB      = ..\slprj\ert\_sharedutils\rtwshared.lib  
MODELLIB        = topmodellib.lib  
MODELREF_LINK_LIBS = submodel1_rtwlib.lib submodel2_rtwlib.lib
```

Recompiling Precompiled Libraries

You can recompile precompiled libraries included as part of the Real-Time Workshop 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 “Precompiling S-Function Libraries” on page 32-99.

Customizing Post Code Generation Build Processing

The Real-Time Workshop 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 appropriate for your application.
- 4** Build the model.

- 5 Modify the command, if necessary, and rebuild the model. Repeat this step until the build results are acceptable.

Build Information Object

At the start of a model build, the Real-Time Workshop 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 “Alphabetical List” in the Real-Time Workshop documentation. “Programming a Post Code Generation Command” on page 27-15 explains how to use the functions to control post code generation build processing.

Programming a Post Code Generation Command

For certain applications, it might be necessary to control aspects of the build process after the code generation. For example, this is necessary when 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 Real-Time Workshop build process cannot find a function you use in your command, the build process errors out.

You can then call any combination of build information functions 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 filenames and paths of the source and include files generated for a model for analysis.

```
function analyzegencode(buildInfo)
% Get the names and paths of all 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.
.
```

For a list of available functions and detailed function descriptions, see “Alphabetical List” in the Real-Time Workshop documentation.

Defining a Post Code Generation Command

After you program a post code generation command, you need to inform the Real-Time Workshop 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 Real-Time Workshop 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',...
'pcgFunctionName');
```

```
pcgFunctions);
```

The following call to `set_param` defines `PostCodeGenCommand` to evaluate the function `analyzegencode`.

```
set_param(model, 'PostCodeGenCommand',...
'analyzegencode(buildInfo));
```

Suppressing Makefile Generation

The Real-Time Workshop 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 Real-Time Workshop build process to not generate a makefile, do one of the following:

- Clear the **Generate makefile** option on the **Real-Time Workshop** pane of the Configuration Parameters dialog box.
- Set the value of the configuration parameter `GenerateMakefile` to `off`.

When you suppress makefile generation,

- You no longer can explicitly specify a make command or template makefile.
- You must specify your own instructions for any post code generation processing, including compilation and linking, in a post code generation command as explained in “Programming a Post Code Generation Command” on page 27-15 and “Defining a Post Code Generation Command” on page 27-17.

Configuring Generated Code with TLC

In this section...

“Introduction” on page 27-19

“Assigning Target Language Compiler Variables” on page 27-19

“Setting Target Language Compiler Options” on page 27-21

Introduction

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 **Real-Time Workshop** pane on the Configuration Parameters dialog. There are two ways to set TLC variables and options, as described in this section.

Note You should not customize TLC files in the directory `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 the Target Language Compiler documentation. 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	<p>When the Enable local block outputs check box is selected, the total allocation size of local variables that are declared by all block outputs in the model cannot exceed MaxStackSize (in bytes). MaxStackSize can be any 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 <code>rtInf</code>, that is, unlimited stack size.</p> <p>Note: Local variables in the generated code from sources other than local block outputs, such as from a Stateflow diagram or Embedded 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.</p>
MaxStackVariableSize=N	<p>When the Enable local block outputs check box is selected, this limits the size of any 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.</p>
WarnNonSaturatedBlocks=value	<p>Flag to control display of overflow warnings for blocks that have saturation capability, but have it turned off (unchecked) in their dialog. These are the options:</p> <ul style="list-style-type: none"> • 0 — No warning is displayed. • 1 — Displays one warning for the model during code generation • 2 — Displays one warning that contains a list of all offending blocks

For more information, see the Target Language Compiler documentation.

Setting Target Language Compiler Options

You can enter TLC options in the **TLC options** edit field on the **Real-Time Workshop** pane of the Configuration Parameters dialog. For information about these options see “Specifying TLC Options” on page 25-11 and the Target Language Compiler documentation.

Customizing the Target Build Process with the STF_make_rtw Hook File

In this section...

- “Overview” on page 27-22
- “File and Function Naming Conventions” on page 27-22
- “STF_make_rtw_hook.m Function Prototype and Arguments” on page 27-23
- “Applications for STF_make_rtw_hook.m” on page 27-27
- “Using STF_make_rtw_hook.m for Your Build Procedure” on page 27-28

Overview

The build process lets you 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.

This section describes an important MATLAB hook, 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 hook 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 correct *hookMethod* argument (as well as other arguments described below). You need to implement only those hook methods that your build process requires.

File and Function Naming Conventions

To ensure that *STF_make_rtw_hook* is called correctly by the build process, you must ensure that the following conditions are met:

- The *STF_make_rtw_hook.m* file is on the MATLAB path.
- The filename is the name of your system target file (*STF*), appended to the string *_make_rtw_hook.m*. For example, if you were generating code with a custom system target file *mytarget.tlc*, you would name your

STF_make_rtw_hook.m file to *mytarget_make_rtw_hook.m*. Likewise, the hook function implemented within the file should follow the same naming convention.

- The hook function implemented in the file follows the function prototype described in the next section.

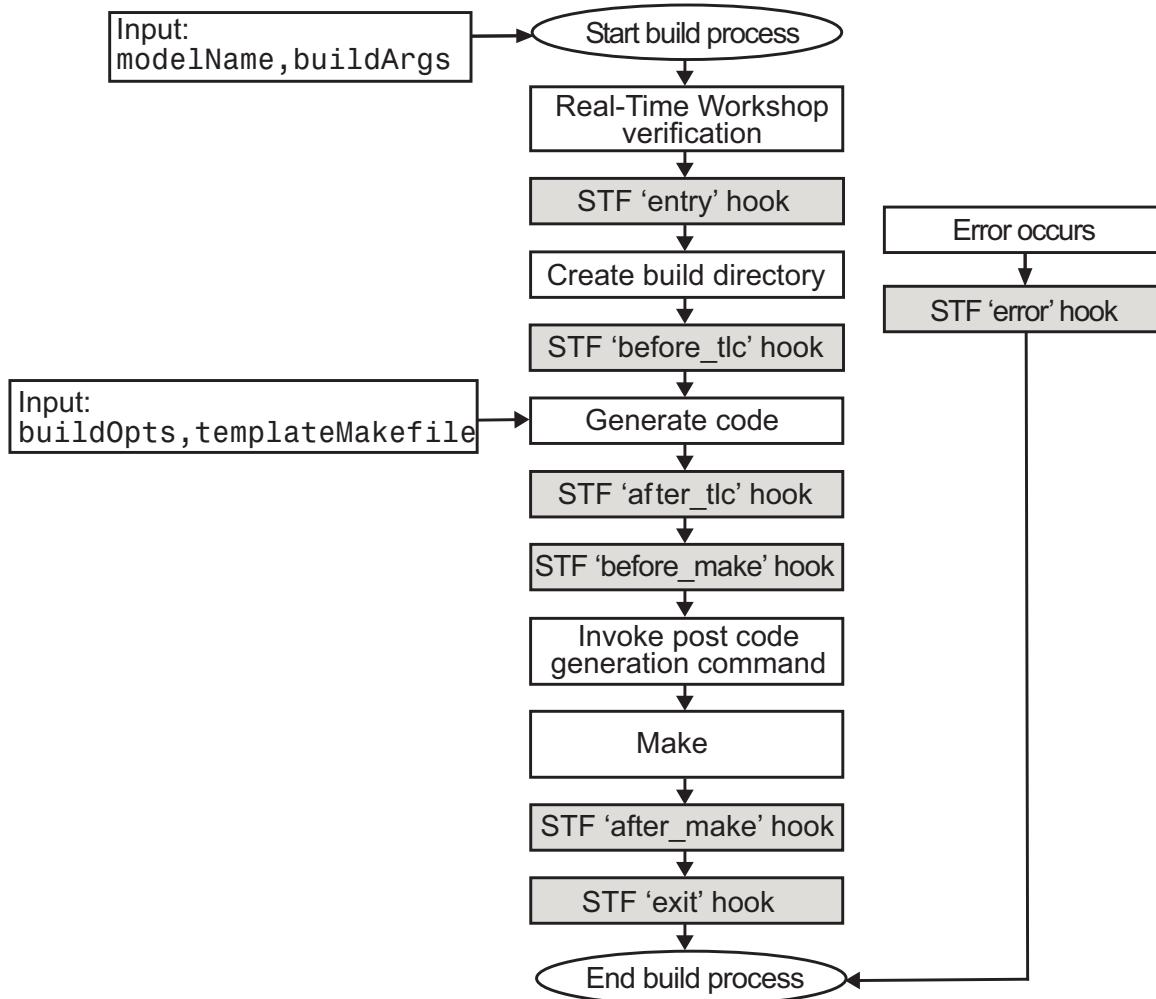
STF_make_rtw_hook.m Function Prototype and Arguments

The function prototype for *STF_make_rtw_hook* is

```
function STF_make_rtw_hook(hookMethod, modelName, rtwRoot, templateMakefile,  
buildOpts, buildArgs)
```

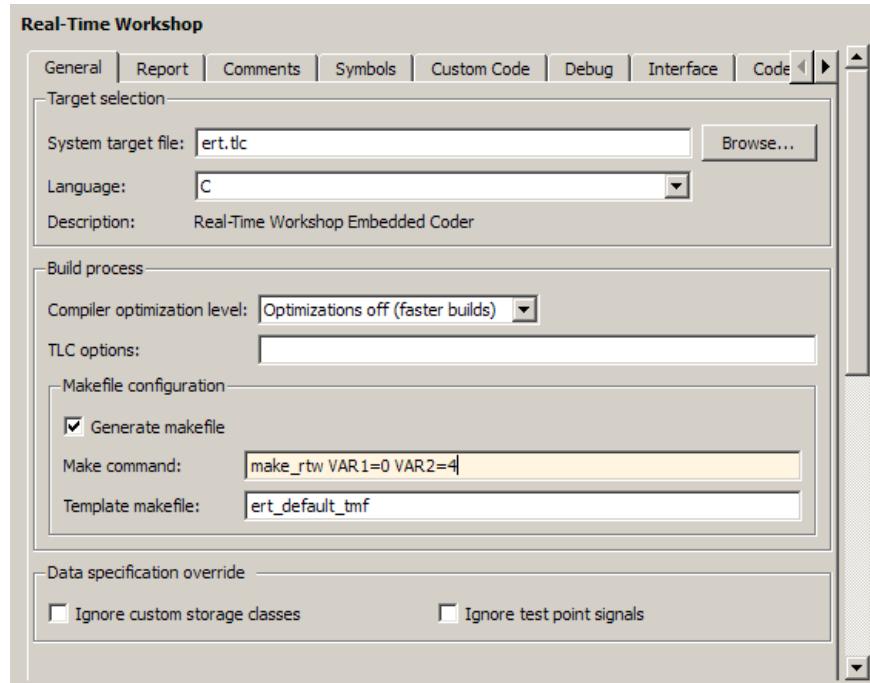
The arguments are defined as:

- **hookMethod**: String specifying the stage of build process from which the *STF_make_rtw_hook* function is called. The flowchart below 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 list below. Valid for the 'before_make', 'after_make', and 'exit' stages only. The buildOpts fields are
 - **modules**: Character array specifying a list of additional files that need to be compiled.
 - **codeFormat**: Character array containing code format specified for the target. (ERT-based targets must use the 'Embedded-C' code format.)
 - **noninlinedSFcns**: Cell array specifying list of noninlined S-functions in the model.
 - **compilerEnvVal**: String specifying 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 (if any) following "make_rtw" in the **Make command** field of the **Real-Time Workshop** pane of the Configuration Parameters dialog box.



The make arguments from the **Make command** field in the figure above, for example, generate the following:

```
% make -f untitled.mk VAR1=0 VAR2=4
```

Applications for STF_make_rtw_hook.m

An enumeration of all possible uses for *STF_make_rtw_hook.m* is beyond the scope of this document. However, this section provides some suggestions of how you might apply the available hooks.

In general, you can use the '*entry*' hook to initialize the build process before any code is generated, for example to change or validate settings. 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 any stage after '*entry*' to obtain the path to the build directory. At the '*exit*' stage, you could then locate generated code files within the build directory 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 that the build process temporarily changes the MATLAB working directory to the build directory for stages '*before_make*', '*after_make*', '*exit*', and '*error*'. Your *STF_make_rtw_hook.m* file should not make incorrect assumptions about the location of the build directory. You can obtain the path to the build directory anytime after the '*entry*' stage. In the following code example, the build directory path is returned as a string to the variable *buildDirPath*.

```
makertwObj = get_param(gcs, 'MakeRTWSettingsObject');
buildDirPath = getfield(makertwObj, 'BuildDirectory');
```

Using `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/rtw/targets/encoder` directory) as follows:

- 1** Copy `ert_make_rtw_hook.m` to a directory in the MATLAB path, and rename it in accordance with the naming conventions described in “File and Function Naming Conventions” on page 27-22. 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 filename.
- 3** Implement the hooks that you require by adding code to the appropriate case statements within the `switch hookMethod` statement.

Customizing the Target Build Process with `sl_customization.m`

In this section...

“Overview” on page 27-29

“Registering Build Process Hook Functions Using `sl_customization.m`” on page 27-31

“Variables Available for `sl_customization.m` Hook Functions” on page 27-32

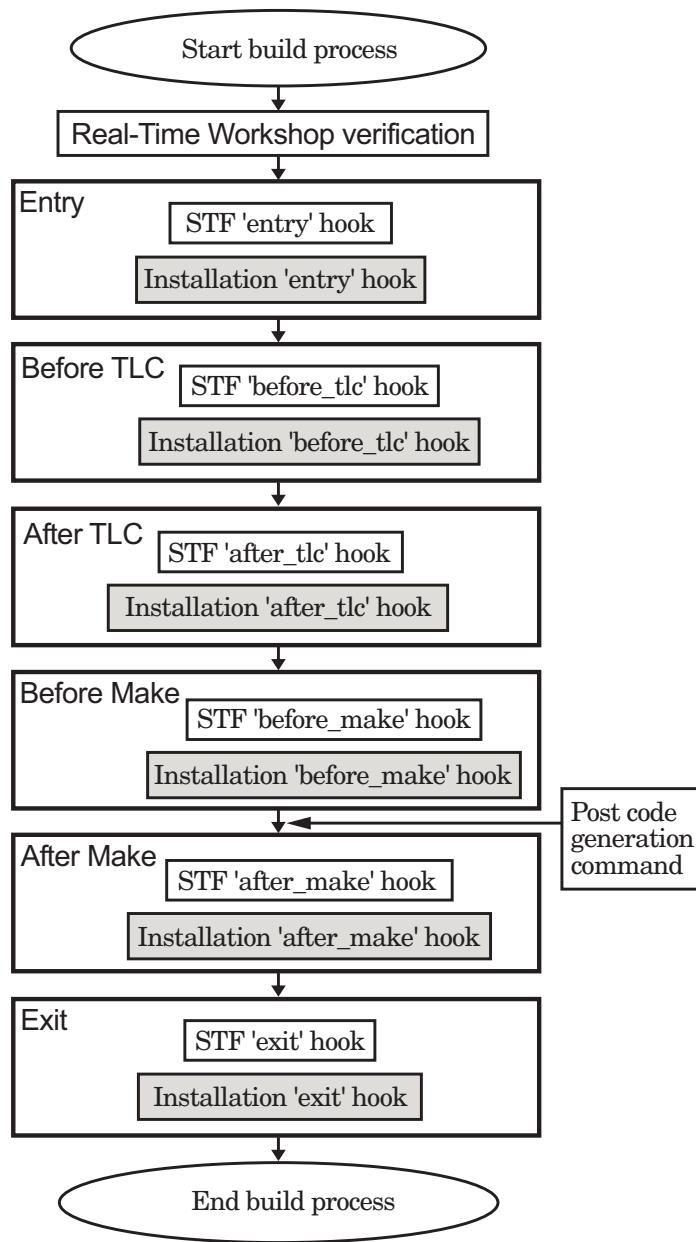
“Example Build Process Customization Using `sl_customization.m`” on page 27-32

Overview

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 “Customizing the Simulink User Interface” in the Simulink documentation.

The `sl_customization.m` file can be used to register installation-specific hook functions to be invoked during the Real-Time Workshop build process. The hook functions that you register through `sl_customization.m` complement System Target File (STF) hooks (described in “Customizing the Target Build Process with the `STF_make_rtw` Hook File” on page 27-22) and post-code generation commands (described in “Customizing Post Code Generation Build Processing” on page 27-14).

The following figure shows the relationship between installation-level hooks and the other available mechanisms for customizing the build process.



Registering Build Process Hook Functions Using `sl_customization.m`

To register installation-level hook functions that will be invoked during the Real-Time Workshop build process, you create an 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 Real-Time Workshop 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 at the MATLAB command line to effect 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 Real-Time Workshop® Build Process Customizations on page 27-32 uses the `addUserHook` method to specify installation-specific build process hooks to be invoked at the '`entry`' and '`after_tlc`' stages of the Real-Time Workshop build. For the hook function source code, see Example 2: `CustomRTWEEntryHook.m` on page 27-33 and Example 3: `CustomRTWPostProcessHook.m` on page 27-33.

Example 1: `sl_customization.m` for Real-Time Workshop Build Process Customizations

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

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

% Register Real-Time Workshop build process hooks
hObj.addUserHook('entry', 'CustomRTWEEntryHook(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 Real-Time Workshop builds. For example, if you open the ERT-based model `rtwdemo_udt`, open the **Real-Time Workshop** pane of the Configuration Parameters dialog box, and click the **Build** button to initiate a Real-Time Workshop build, the following messages are displayed:

```
>> rtwdemo_udt

### Starting Real-Time Workshop build procedure for model: rtwdemo_udt
Custom entry hook for model 'rtwdemo_udt.'
Custom post process hook for model 'rtwdemo_udt.'
### Successful completion of Real-Time Workshop build procedure for model: rtwdemo_udt
>>
```

Replacing 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 all properties that were formerly specified in your *STF_rtw_info_hook* file.

For backward compatibility, existing *STF_rtw_info_hook* files continue to operate correctly. However, you should convert your target and models to use of the **Hardware Implementation** pane. See “Configuring the Hardware Implementation” on page 33-2 .

Converting MATLAB Code to C/C++ Code

- “About Embedded MATLAB Coder” on page 28-3
- “Workflows for Converting MATLAB Code to C/C++ Code” on page 28-5
- “Installing Prerequisite Products for Embedded MATLAB Coder” on page 28-8
- “Setting Up the C/C++ Compiler” on page 28-9
- “File Paths and Naming Conventions” on page 28-11
- “Making MATLAB Code Compliant with the Embedded MATLAB Subset” on page 28-14
- “Configuring Your Environment for Code Generation” on page 28-16
- “Specifying Properties of Primary Function Inputs” on page 28-23
- “Choosing Your Target” on page 28-41
- “Compiling Your MATLAB File” on page 28-46
- “Compiling More Than One Entry-Point MATLAB Function” on page 28-48
- “Generating C/C++ Code from MATLAB Code That Uses Global Data” on page 28-52
- “How `emlc` Generates Code” on page 28-59
- “Working with Compilation Reports” on page 28-70
- “Calling Generated C/C++ Functions” on page 28-88
- “Integrating Custom C/C++ Code with Generated Code” on page 28-95

- “Speeding Up Simulation with the Basic Linear Algebra Subprograms (BLAS) Library” on page 28-98
- “Controlling Runtime Checks” on page 28-101
- “Debugging Runtime Errors” on page 28-104

About Embedded MATLAB Coder

In this section...

“Converts MATLAB Code to C/C++ Code” on page 28-3

“Running a Demo of Embedded MATLAB Coder” on page 28-4

“How Embedded MATLAB Coder Resolves Function Calls” on page 28-4

Converts MATLAB Code to C/C++ Code

Embedded MATLAB Coder works with the Real-Time Workshop software to automatically convert MATLAB code to C/C++ code. With the Embedded MATLAB Coder, you can:

Package generated C/C++ code as an executable, library, or MEX function

MEX-files are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute. For more information, see “Using MEX-Files to Call C/C++ and Fortran Programs” in the MATLAB External Interfaces documentation.

Verify that your MATLAB code complies with the Embedded MATLAB syntax and semantics.

For more information, see “Making MATLAB Code Compliant with the Embedded MATLAB Subset” on page 28-14.

Accelerate MATLAB code that uses Fixed-Point Toolbox functions.

The generated code contains optimizations that automatically accelerate fixed-point arithmetic operations.

Generate C/C++ code from source-protected P-code

For more information, see `eml.allowpcode` in the *Embedded MATLAB Reference*.

Note For information about Embedded MATLAB language semantics and syntax, see “Working with the Embedded MATLAB Subset” in the Embedded MATLAB documentation.

Running a Demo of Embedded MATLAB Coder

The Real-Time Workshop product ships with a demo of how to generate embeddable C code from MATLAB code. The demo uses standard MATLAB library functions to detect edges in an image. As part of the demo, you can compile, run, and view the generated C code, and display the results.

If you have a Real-Time Workshop license, you can open the demo using the following MATLAB command:

```
>> showdemo rtwdemo_emlcbasicdemo
```

How Embedded MATLAB Coder Resolves Function Calls

Embedded MATLAB Coder resolves function calls by first searching the Embedded MATLAB path and then the MATLAB path. By default, Embedded MATLAB Coder tries to compile and generate code for functions it finds on the path unless you explicitly declare the function to be extrinsic. Embedded MATLAB Coder does not compile extrinsic functions, but rather dispatches them to the MATLAB interpreter for execution. For more information, see “How the Embedded MATLAB Subset Resolves Function Calls” in the Embedded MATLAB documentation.

To learn about calling MATLAB functions from Embedded MATLAB code, see “Calling MATLAB Functions” in the Embedded MATLAB documentation.

Workflows for Converting MATLAB Code to C/C++ Code

In this section...

“Workflow for Converting MATLAB Code to Embeddable C/C++ Code” on page 28-5

“Workflow for Converting MATLAB Code to a MEX Function” on page 28-6

Workflow for Converting MATLAB Code to Embeddable C/C++ Code

Follow these steps for converting your MATLAB code to an embeddable C/C++ executable or library:

Step	Action	Details
1	Install prerequisite products.	See “Installing Prerequisite Products for Embedded MATLAB Coder” on page 28-8.
2	Set up your C/C++ compiler.	See “Setting Up the C/C++ Compiler” on page 28-9.
3	Set up your file infrastructure based on compile search paths and naming conventions.	See “File Paths and Naming Conventions” on page 28-11.
4	Make your MATLAB code compliant with the Embedded MATLAB subset.	See “Making MATLAB Code Compliant with the Embedded MATLAB Subset” on page 28-14.
5	Configure your environment for code generation.	See “Configuring Your Environment for Code Generation” on page 28-16.
6	Specify properties of primary function inputs.	See “Specifying Properties of Primary Function Inputs” on page 28-23.
7	Choose your target.	See “Choosing Your Target” on page 28-41.

Step	Action	Details
8	Run Embedded MATLAB Coder with the appropriate command line options.	<p>At a minimum, you must specify:</p> <ul style="list-style-type: none"> • <code>-T rtw:exe</code> to generate a C/C++ code executable or <code>-T rtw:lib</code> to generate a C/C++ code library • A .c or .cpp file that contains the main function to use for generating a C/C++ executable <p>See “Compiling Your MATLAB File” on page 28-46.</p>
9	Generate and interpret compilation reports.	See “Working with Compilation Reports” on page 28-70.
10	Modify your code as necessary to meet special requirements for calling generated C/C++ functions from your application.	See “Calling Generated C/C++ Functions” on page 28-88.

Workflow for Converting MATLAB Code to a MEX Function

Follow these steps for converting your MATLAB code to a MEX function:

Step	Action	Details
1	Install prerequisite products.	See “Installing Prerequisite Products for Embedded MATLAB Coder” on page 28-8.
2	Set up your C/C++ compiler.	See “Setting Up the C/C++ Compiler” on page 28-9.
3	Set up your file infrastructure based on compile search paths and naming conventions.	See “File Paths and Naming Conventions” on page 28-11.
4	Make your MATLAB code compliant with the Embedded MATLAB subset.	See “Making MATLAB Code Compliant with the Embedded MATLAB Subset” on page 28-14.

Step	Action	Details
5	Specify properties of primary function inputs.	See “Specifying Properties of Primary Function Inputs” on page 28-23.
6	Run Embedded MATLAB Coder with the appropriate command-line options.	See “Compiling Your MATLAB File” on page 28-46.

Installing Prerequisite Products for Embedded MATLAB Coder

Embedded MATLAB Coder ships with the Real-Time Workshop product. To use Embedded MATLAB Coder, you must install the Real-Time Workshop product and its prerequisite products:

- MATLAB
- Simulink
- C/C++ Compiler

For instructions on installing The MathWorks products, see the MATLAB installation documentation for your platform. Enter the `ver` command in the MATLAB Command Window to check what other MathWorks products are installed.

For instructions on installing and setting up a C/C++ compiler, see “Setting Up the C/C++ Compiler” on page 28-9.

Setting Up the C/C++ Compiler

In this section...

“How to Set Up Your C/C++ Compiler” on page 28-9

“Supported Compilers for Generating MEX Functions” on page 28-9

“Supported Compilers for Generating C/C++ Code” on page 28-9

How to Set Up Your C/C++ Compiler

Before using Embedded MATLAB Coder, you must set up your C/C++ compiler by running the `mex -setup` command, as described in the documentation for `mex` in the MATLAB Function Reference. You must run this command even if you use the default C compiler that comes with the MATLAB product for Microsoft Windows platforms. You can also use `mex` to choose and configure a different C/C++ compiler, as described in “Building MEX-Files” in the MATLAB External Interfaces documentation.

Supported Compilers for Generating MEX Functions

Embedded MATLAB Coder supports the following MEX compilers:

- Lcc-win32 C 2.4.1
- Microsoft Visual C++ 2008 Express
- Microsoft Visual C++ 2005
- Microsoft Visual C++ 6.0
- Open WATCOM C++ 1.7

Supported Compilers for Generating C/C++ Code

For generating C/C++ code, Embedded MATLAB Coder supports the same C/C++ compilers as Simulink supports for code generation with Real-Time Workshop. For a list of these compilers:

- 1 Navigate to the Supported and Compatible Compilers Web site.
- 2 Select your platform.

- 3** In the table for Simulink and related products, find the compilers checked in the column titled Real-Time Workshop.

File Paths and Naming Conventions

In this section...

- “Compile Path Search Order” on page 28-11
- “Can I Add Files to the Embedded MATLAB Path?” on page 28-11
- “When to Use the Embedded MATLAB Path” on page 28-11
- “Adding Folders to Search Paths” on page 28-12
- “Naming Conventions” on page 28-12

Compile Path Search Order

Embedded MATLAB Coder resolves MATLAB functions by searching first on the Embedded MATLAB path and then on the MATLAB path. See “How the Embedded MATLAB Subset Resolves Function Calls” in the Embedded MATLAB documentation.

Can I Add Files to the Embedded MATLAB Path?

With Embedded MATLAB Coder, you can prepend folders and files to the Embedded MATLAB path, as described in “Adding Folders to Search Paths” on page 28-12. By default, the Embedded MATLAB path contains the current folder and the Embedded MATLAB libraries.

When to Use the Embedded MATLAB Path

Use the Embedded MATLAB path to override a MATLAB function with a customized version. Because Embedded MATLAB Coder searches the Embedded MATLAB path first, a MATLAB file on the Embedded MATLAB path always shadows a MATLAB file of the same name on the MATLAB path. To override a MATLAB function with a version implemented in Embedded MATLAB libraries, follow these steps:

- 1 Create each version of the MATLAB function in identically-named files.
- 2 Add the MATLAB version of the function to the MATLAB path.

- 3** Add the Embedded MATLAB version of the function to the Embedded MATLAB path.

See “Adding Folders to Search Paths” on page 28-12.

Adding Folders to Search Paths

The following table explains how to add folders to search paths:

To add folders to:	Do this:
Embedded MATLAB path	Prepend folders to the Embedded MATLAB path using the compiler option -I. See “Adding Folders to Search Paths” using <code>emlc</code> in the Real-Time Workshop Function Reference.
MATLAB path	Follow the instructions in “Adding a Folder to the Search Path” in the MATLAB Programming Fundamentals documentation.

Naming Conventions

Embedded MATLAB Coder enforces naming conventions for MATLAB functions and generated files.

- “Reserved Prefixes” on page 28-12
- “Conventions for Naming Generated files” on page 28-12

Reserved Prefixes

Embedded MATLAB Coder reserves the prefix `eml` for global C/C++ functions and variables in generated code. For example, Embedded MATLAB runtime library function names all begin with the prefix `emlrt`, such as `emlrtCallMATLAB`. To avoid naming conflicts, do not name C/C++ functions or primary MATLAB functions with the prefix `eml`.

Conventions for Naming Generated files

The following table describes how Embedded MATLAB Coder names generated files. Embedded MATLAB Coder follows MATLAB conventions by providing platform-specific extensions for MEX files.

Platform	MEX File Extension	Real-Time Workshop Library Extension	Real-Time Workshop Executable Extension
Linus Torvalds' Linux (32-bit)	.mexglx	.a	None
Linux x86-64	.mexa64	.a	None
Microsoft Windows (32-bit)	.mexw32	.lib	.exe
Windows x64	.mexw64	.lib	.exe

Making MATLAB Code Compliant with the Embedded MATLAB Subset

In this section...

“Debugging Strategies” on page 28-14

“Detecting Embedded MATLAB Syntax Violations at Compile Time” on page 28-15

Debugging Strategies

Before you perform code verification, The MathWorks recommends that you choose a debugging strategy for detecting and correcting noncompliant code in your MATLAB applications, especially if they consist of a large number of MATLAB files that call each other’s functions. Here are two best practices:

Debugging Strategy	What to Do	Pros	Cons
Bottom-up verification	<ul style="list-style-type: none"> 1 Verify that your lowest-level (leaf) functions are compliant. 2 Work your way up the function hierarchy incrementally to compile and verify each function, ending with the top-level function. 	<ul style="list-style-type: none"> Efficient Safe Easy to isolate Embedded MATLAB syntax violations 	Requires application tests that work from the bottom up
Top-down verification	<ul style="list-style-type: none"> 1 Declare all functions called by the top-level function to be extrinsic so Embedded MATLAB Coder does not compile them (see “Declaring MATLAB Functions as Extrinsic Functions” in 	Lets you retain your top-level tests	Introduces extraneous code that you must remove after code verification, including: <ul style="list-style-type: none"> Extrinsic declarations Additional assignment statements as necessary to convert opaque values returned by extrinsic functions to nonopaque

Debugging Strategy	What to Do	Pros	Cons
	<p>the Embedded MATLAB documentation).</p> <p>2 Verify that your top-level function is compliant.</p> <p>3 Work your way down the function hierarchy incrementally by removing extrinsic declarations one by one to compile and verify each function, ending with the leaf functions.</p>		values (see “Working with mxArrays” in the Embedded MATLAB documentation)

Detecting Embedded MATLAB Syntax Violations at Compile Time

Before you can successfully convert a MATLAB file to C/C++ code, you must verify that your MATLAB code complies with Embedded MATLAB syntax and semantics, as defined in “Working with the Embedded MATLAB Subset”.

Embedded MATLAB Coder checks for all potential Embedded MATLAB syntax violations at compile time. When Embedded MATLAB Coder detects errors or warnings, it automatically generates a compilation report that describes the issues and provides links to the offending MATLAB code. See “Working with Compilation Reports” on page 28-70.

If your MATLAB code calls functions on the MATLAB path, Embedded MATLAB Coder attempts to compile these functions unless you declare them to be extrinsic (see “How the Embedded MATLAB Subset Resolves Function Calls” in the Embedded MATLAB documentation). To get detailed diagnostics, add the %#eml compiler directive to each external function that you want Embedded MATLAB Coder to compile, as described in “Adding the Compilation Directive %#eml” in the Embedded MATLAB documentation.

Configuring Your Environment for Code Generation

In this section...

- “Types of Configuration Objects” on page 28-16
- “Working with Configuration Objects” on page 28-18
- “Creating Configuration Objects” on page 28-19
- “Modifying Configuration Objects” on page 28-20
- “Saving Configuration Objects” on page 28-21

Types of Configuration Objects

Embedded MATLAB Coder provides configuration objects for customizing your environment for code generation. You can create the following configuration objects:

Configuration Object	Description	Default Target	Parameter Reference
emlcoder.MEXConfig	Specifies parameters for MEX code generation (for MEX builds only)	mex (generates MEX code unless you specify a different target)	See “Automatic C MEX Generation Dialog Box for Embedded MATLAB Coder” in the Real-Time Workshop Reference documentation.

Configuration Object	Description	Default Target	Parameter Reference
<p><code>emlcoder.CompilerOptions</code></p> <p>Note This configuration object is for use with the Embedded MATLAB function <code>emlmex</code>. It is available for use with <code>emlc</code> for backwards compatibility purposes only and will be removed in a future release. To specify parameters for embeddable C/C++ code generation using Embedded MATLAB Coder, use the MEX or Real-Time Workshop configuration objects.</p>	<p>Specifies parameters for fine-tuning compilation.</p>		<p>See “Compiler Options Dialog Box” in the Real-Time Workshop Reference documentation.</p>
<p><code>emlcoder.RTWConfig</code> or <code>emlcoder.RTWConfig('grt')</code></p> <p>Creates a Real-Time Workshop configuration object for a GRT-based target (default).</p> <p><code>emlcoder.RTWConfig('ert')</code></p> <p>Creates a Real-Time Workshop configuration object for an ERT-based target.</p> <p>Note ERT-based targets require a Real-Time Workshop Embedded Coder license when generating code.</p>	<p>Specifies parameters for embeddable C/C++ code generation (for Real-Time Workshop builds only).</p>	<p><code>rtw:lib</code> (generates an embeddable C library unless you specify a different target and language)</p>	<p>See “Real-Time Workshop Dialog Box for Embedded MATLAB Coder” in the Real-Time Workshop Reference documentation.</p>
<p><code>emlcoder.HardwareImplementation</code></p>	<p>Specifies parameters of the target</p>		<p>See “Hardware Implementation Dialog Box</p>

Configuration Object	Description	Default Target	Parameter Reference
	hardware implementation (for Real-Time Workshop builds only). If not specified, Embedded MATLAB Coder generates code that is compatible with the MATLAB host computer.		for Embedded MATLAB Coder” in the Real-Time Workshop Reference documentation.

For more information about the association between configuration objects and targets, see “Choosing Your Target” on page 28-41.

Working with Configuration Objects

To use configuration objects to customize your environment for code generation, follow these steps:

- 1 Define configuration object variables in the MATLAB workspace, as described in “Creating Configuration Objects” on page 28-19.

For example, the following command defines a Real-Time Workshop configuration object variable called `rtwcfg`:

```
% To create a configuration object for a GRT-based target
rtwcfg=emlcoder.RTWConfig;

% To create a configuration object for an ERT-based target
% This requires a Real-Time Workshop Embedded Coder license
% when generating code
rtwcfg=emlcoder.RTWConfig('ert');
```

- 2 Modify the parameters of the configuration object as necessary, using one of these methods:

- Interactive commands, as described in “Modifying Configuration Objects at the Command Line Using Dot Notation” on page 28-21
 - Graphical user interface, as described in “Modifying Configuration Objects Using Dialog Boxes” on page 28-21
- 3** Invoke Embedded MATLAB Coder with the `-s` option and specify the configuration object as its argument.

The `-s` option instructs Embedded MATLAB Coder to generate code for the target, based on the configuration property values. In the following example, Embedded MATLAB Coder generates a C/C++ code library from a MATLAB file `myMfile.m`, based on the parameters of a Real-Time Workshop configuration object `rtwcfg` defined in the first step:

```
emlc -T rtw:lib -s rtwcfg myMfile
```

Note that the `-T` option specifies the type of target you want to build — in this case, a library of embeddable C/C++ code. For more information about invoking Embedded MATLAB Coder, see `emlc` in the Real-Time Workshop Function Reference.

Creating Configuration Objects

You can define a configuration object in the MATLAB workspace, as follows:

To Create:	Use a Constructor Command Like This
MEX configuration object	<code>mexcfg = emlcoder.MEXConfig</code>
Real-Time Workshop configuration object for generating embeddable C/C++ code for a GRT-based target (default)	<code>rtwcfg = emlcoder.RTWConfig</code> or <code>rtwcfg = emlcoder.RTWConfig('grt')</code>
Real-Time Workshop configuration object for generating embeddable C/C++ code for an ERT-based target	<code>rtwcfg = emlcoder.RTWConfig('ert')</code>

To Create:	Use a Constructor Command Like This
Hardware implementation configuration object	<code>hwcfg = emlcoder.HardwareImplementation</code>
Compiler options configuration object	<code>cocfg = emlcoder.CompilerOptions</code>

Note This configuration object is for use with the Embedded MATLAB function `emlmex`. It is available for use with `emlc` for backwards compatibility purposes only and will be removed in a future release. To specify parameters for embeddable C/C++ code generation using Embedded MATLAB Coder, use the MEX or Real-Time Workshop configuration objects.

Each configuration object comes with a set of parameters, initialized to default values (see “Configuration Parameters for Embedded MATLAB Coder” in the Real-Time Workshop Reference documentation). You can change these settings as described in “Modifying Configuration Objects” on page 28-20.

Modifying Configuration Objects

There are two ways to modify the values of configuration objects for Embedded MATLAB Coder.

- “Modifying Configuration Objects at the Command Line Using Dot Notation” on page 28-21
- “Modifying Configuration Objects Using Dialog Boxes” on page 28-21

Modifying Configuration Objects at the Command Line Using Dot Notation

You can use dot notation to modify the value of one configuration object parameter at a time, using this syntax:

```
configuration_object.property = value
```

Dot notation uses assignment statements to modify configuration object properties:

- To specify a `main` function during C/C++ code generation, enter this code at the command line:

```
rtwcfg = emlcoder.RTWConfig
% Configuration object for a GRT-based target
rtwcfg.CustomInclude = 'c:\myfiles';
rtwcfg.CustomSource = 'main.c';
emlc -T rtw:exe -s rtwcfg myFun
```

- To automatically generate and launch HTML reports after generating a C/C++ library, enter this code:

```
rtwcfg = emlcoder.RTWConfig
% Configuration object for a GRT-based target
rtwcfg.GenerateReport= true
rtwcfg.LaunchReport = true
emlc -T rtw:lib -s rtwcfg myFun
```

Modifying Configuration Objects Using Dialog Boxes

Besides using commands, you can modify code generation parameters using dialog boxes. To learn how to work with these dialog boxes, see “Configuration Parameters for Embedded MATLAB Coder” in the Real-Time Workshop Reference documentation.

Saving Configuration Objects

Configuration objects do not automatically persist between MATLAB sessions, but there are two ways to preserve your settings:

Save a configuration object to a MAT-file and then load the MAT-file at your next session

For example, assume you create and customize a MEX configuration object `mexcfg` in the MATLAB workspace. To save the configuration object, type this command at the MATLAB prompt:

```
save mexcfg.mat mexcfg
```

The `save` command saves `mexcfg` to the file `mexcfg.mat` in the current folder.

To restore `mexcfg` in a new MATLAB session, type this command at the MATLAB prompt:

```
load mexcfg.mat
```

The `load` command loads the objects defined in `mexcfg.mat` to the MATLAB workspace.

Write a script that creates the configuration object and sets its properties.

You can rerun the script whenever you need to use the configuration object again.

Specifying Properties of Primary Function Inputs

In this section...

- “Why You Must Specify Input Properties” on page 28-23
- “Properties to Specify” on page 28-23
- “Rules for Specifying Properties of Primary Inputs” on page 28-27
- “Methods for Defining Properties of Primary Inputs” on page 28-27
- “Defining Input Properties by Example at the Command Line” on page 28-28
- “Defining Input Properties Programmatically in the MATLAB File” on page 28-31

Why You Must Specify Input Properties

Because C and C++ are statically typed languages, Embedded MATLAB Coder must determine the properties of all variables in the MATLAB files at compile time. To infer variable properties in MATLAB files, Embedded MATLAB Coder must be able to identify the properties of the inputs to the *primary* function, also known as the *top-level* or *entry-point* function. Therefore, if your primary function has inputs, you must specify the properties of these inputs, also called *preconditions*, to Embedded MATLAB Coder. If your primary function has no input parameters, Embedded MATLAB Coder can compile your MATLAB file without modification. You do not need to specify properties of inputs to subfunctions or external functions called by the primary function.

Properties to Specify

If your primary function has inputs, you must specify the following properties for each input:

For:	Specify Properties:				
	Class	Size	Complexity	numerictype	fimath
Fixed-point inputs	✓	✓	✓	✓	✓

For:	Specify Properties:				
Each field in a structure input	<p>Specify properties for each field according to its class</p> <p>When a primary input is a structure, Embedded MATLAB Coder treats each field as a separate input. Therefore, you must specify properties for all fields of a primary structure input in the order that they appear in the structure definition, as follows:</p> <ul style="list-style-type: none"> • For each field of input structures, specify class, size, and complexity. • For each field that is fixed-point class, also specify <code>numerictype</code>, and <code>fimath</code>. 				
All other inputs	✓	✓	✓		

Default Property Values

Embedded MATLAB Coder assigns the following default values for properties of primary function inputs:

Property	Default
<code>class</code>	<code>double</code>
<code>size</code>	<code>scalar</code>
<code>complexity</code>	<code>real</code>
<code>numerictype</code>	No default
<code>fimath</code>	MATLAB default <code>fimath</code> object

Specifying Default Values for Structure Fields. In most cases, Embedded MATLAB Coder uses defaults when you don't explicitly specify values for properties — except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore, you may need to specify default values for properties of structure fields. For examples, see “Example: Specifying Class and Size of Scalar Structure” on page 28-38 and “Example: Specifying Class and Size of Structure Array” on page 28-39.

Specifying Default fimath Values for MEX Functions. MEX functions generated with Embedded MATLAB Coder use the default `fimath` value in effect at compile time. You can set the default `fimath` value explicitly for functions that use fixed-point data by specifying the “-F Specify Default `fimath`” option with the `emlc` command. If you do not specify a default `fimath` value on the command line, Embedded MATLAB Coder uses the MATLAB default `fimath`.

When running MEX functions that depend on the MATLAB default `fimath` value, do not change this value during your MATLAB session. Otherwise, you receive a runtime error, alerting you to a mismatch between the compile-time and runtime `fimath` values.

For example, suppose you define the following MATLAB function `test`:

```
function y = test %#eml
y = fi(0);
```

The function `test` constructs a `fi` object without explicitly specifying a `fimath` object. Therefore, `test` will rely on the default `fimath` object in effect at compile time. At the MATLAB prompt, generate the MEX function `testx` to use the factory setting of the MATLAB default `fimath`:

```
resetdefaultfimath;
emlc -o testx test
```

Next, run `testx` to display the MATLAB default `fimath` value:

```
testx
ans =
0
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 15
```

Now modify the MATLAB default `fimath` value so it no longer matches the factory setting used at compile time:

```
setdefaultfimath(fimath('RoundMode','floor'));
```

Finally, clear the MEX function from memory and rerun it:

```
clear testx
testx
```

The mismatch is detected and causes an error:

```
??? This function was compiled with a different default
fimath than the current default.
```

```
Error in ==> test
```

Supported Classes

The following table presents the class names supported by Embedded MATLAB Coder:

Class Name	Description
<code>logical</code>	Logical array of true and false values
<code>char</code>	Character array
<code>int8</code>	8-bit signed integer array
<code>uint8</code>	8-bit unsigned integer array
<code>int16</code>	16-bit signed integer array
<code>uint16</code>	16-bit unsigned integer array
<code>int32</code>	32-bit signed integer array
<code>uint32</code>	32-bit unsigned integer array
<code>single</code>	Single-precision floating-point or fixed-point number array
<code>double</code>	Double-precision floating-point or fixed-point number array

Class Name	Description
<code>struct</code>	Structure array
<code>embedded.fi</code>	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

Follow these rules when specifying the properties of primary inputs:

- For each primary function input whose class is fixed point (`fi`), you must specify the input's `numerictype` and `fimath` properties.
- For each primary function input whose class is `struct`, you must specify the properties of each of its fields in the order that they appear in the structure definition.

Methods for Defining Properties of Primary Inputs

Use one of the following methods to define the properties of primary function inputs:

Method	Pros	Cons
“Defining Input Properties by Example at the Command Line” on page 28-28	<ul style="list-style-type: none"> • Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a small number of primary inputs 	<ul style="list-style-type: none"> • Must be specified at the command line every time you invoke Embedded MATLAB Coder (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
“Defining Input Properties Programmatically in the MATLAB File” on page 28-31	<ul style="list-style-type: none"> • Integrated with MATLAB code so you do not need to redefine properties each time you invoke Embedded MATLAB Coder • Provides documentation of property specifications in the MATLAB code 	<ul style="list-style-type: none"> • Uses complex syntax

Method	Pros	Cons
	<ul style="list-style-type: none">Efficient for specifying memory-intensive inputs such as large structures	

Note To specify the properties of inputs for any given primary function, use one of these methods, but not both.

Defining Input Properties by Example at the Command Line

- “Command Line Option -eg” on page 28-28
- “Rules for Using the -eg Option” on page 28-29
- “Specifying Constant Inputs Using the -eg Option” on page 28-29
- “Specifying Variable-Size Inputs” on page 28-29
- “Example: Specifying Properties of Primary Inputs by Example” on page 28-29
- “Example: Specifying Properties of Primary Fixed-Point Inputs by Example” on page 28-30

Command Line Option -eg

The command that invokes Embedded MATLAB Coder — `emlc` — provides a command-line option `-eg` for specifying the properties of primary function inputs as a cell array of example values. The cell array can be a variable or literal array of constant values. Using this option, you specify the properties of inputs at the same time as you compile the MATLAB file with Embedded MATLAB Coder.

See “Example: Specifying General Properties of Primary Inputs” on page 28-37 for `emlc` in the Real-Time Workshop Function Reference.

Rules for Using the -eg Option

Follow these rules when using the `-eg` command-line option to define properties by example:

- The cell array of sample values must contain the same number of elements as primary function inputs.
- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature — for example, the first element in the cell array defines the properties of the first primary function input.

Specifying Constant Inputs Using the -eg Option

You can define inputs to be constants using the `-eg` option with `emlc` in the same way as you can with `emlmex`. See “Specifying Constant Inputs” in the *Embedded MATLAB User’s Guide*.

Specifying Variable-Size Inputs

You can define inputs to have one or more variable-size dimensions using the `-eg` option with `emlc` in the same way as you can with `emlmex`. See “Specifying Variable-Size Inputs” in the *Embedded MATLAB User’s Guide*.

Example: Specifying Properties of Primary Inputs by Example

Consider a MATLAB function that adds its two inputs:

```
%#eml
function y = emcf(u,v)
y = u + v;
```

The following examples show how to specify different properties of the primary inputs `u` and `v` by example at the command line:

- Use a literal cell array of constants to specify that both inputs are real scalar doubles:

```
emlc -o emcfx emcf -eg {0,0}
```

- Use a literal cell array of constants to specify that input `u` is an unsigned 16-bit, 1-by-4 vector and input `v` is a scalar double:

```
emlc -o emcfx emcf -eg {zeros(1,4,'uint16'),0}
```

- Assign sample values to a cell array variable to specify that both inputs are real, unsigned 8-bit integer vectors:

```
a = uint8([1;2;3;4])
b = uint8([5;6;7;8])
ex = {a,b}
emlc -o emcfx emcf -eg ex
```

Example: Specifying Properties of Primary Fixed-Point Inputs by Example

Consider an MATLAB function that calculates the square root of a fixed-point number:

```
%#eml
function y = sqrtfi(x)
y = sqrt(x);
```

To specify the properties of the primary fixed-point input x by example on the MATLAB command line, follow these steps:

- 1 Define the `numerictype` properties for x , as in this example:

```
T = numerictype('WordLength',32,
                  'FractionLength',23,
                  'Signed',true);
```

- 2 Define the `fimath` properties for x , as in this example:

```
F = fimath('SumMode','SpecifyPrecision',
            'SumWordLength',32,
            'SumFractionLength',23,
            'ProductMode','SpecifyPrecision',
            'ProductWordLength',32,
            'ProductFractionLength',23);
```

- 3 Create a fixed-point variable with the `numerictype` and `fimath` properties you just defined, as in this example:

```
myeg = { fi(4.0,T,F) };
```

- 4** Compile the function `sqrtfi` using the `emlc` command, passing the variable `myeg` as the argument to the `-eg` option, as in this example:

```
emlc sqrtni -eg myeg;
```

Defining Input Properties Programmatically in the MATLAB File

Embedded MATLAB Coder lets you use the MATLAB `assert` function to define properties of primary function inputs directly in your MATLAB file.

- “How to Use `assert` with Embedded MATLAB Coder” on page 28-31
- “Rules for Using `assert` Function” on page 28-36
- “Example: Specifying General Properties of Primary Inputs” on page 28-37
- “Example: Specifying Properties of Primary Fixed-Point Inputs” on page 28-38
- “Example: Specifying Class and Size of Scalar Structure” on page 28-38
- “Example: Specifying Class and Size of Structure Array” on page 28-39

How to Use `assert` with Embedded MATLAB Coder

Use the `assert` function to invoke standard MATLAB functions for specifying the class, size, and complexity of primary function inputs.

- “Specify Any Class” on page 28-32
- “Specify fi Class” on page 28-32
- “Specify Structure Class” on page 28-33
- “Specify Any Size” on page 28-33
- “Specify Scalar Size” on page 28-34
- “Specify Real Input” on page 28-34
- “Specify Complex Input” on page 28-34
- “Specify numerictype of Fixed-Point Input” on page 28-35
- “Specify fimath of Fixed-Point Input” on page 28-35
- “Specify Multiple Properties of Input” on page 28-36

Specify Any Class.

```
assert ( isa ( param, 'class_name' ) )
```

Sets the input parameter *param* to the MATLAB class *class_name*. For example, to set the class of input U to a 32-bit signed integer, call:

```
...
assert(isa(U,'int32'));
...
```

Note If you set the class of an input parameter to **fi**, you must also set its **numerictype**, see “Specify numerictype of Fixed-Point Input” on page 28-35. You can also set its **fimath** properties, see “Specify fimath of Fixed-Point Input” on page 28-35. If you do not set the **fimath** properties, **emlc** uses the MATLAB default **fimath** value.

If you set the class of an input parameter to **struct**, you must specify the properties of all fields in the order that they appear in the structure definition.

Specify fi Class.

```
assert ( isfi ( param ) )
assert ( isa ( param, 'embedded.fi' ) )
```

Sets the input parameter *param* to the MATLAB class **fi** (fixed-point numeric object). For example, to set the class of input U to **fi**, call:

```
...
assert(isfi(U));
...
```

or

```
...
assert(isa(U,'embedded.fi'));
...
```

Note If you set the class of an input parameter to `fi`, you must also set its `numerictype`, see “Specify numerictype of Fixed-Point Input” on page 28-35. You can also set its `fimath` properties, see “Specify fimath of Fixed-Point Input” on page 28-35. If you do not set the `fimath` properties, `emlc` uses the MATLAB default `fimath` value.

If you set the class of an input parameter to `struct`, you must specify the properties of all fields in the order they appear in the structure definition.

Specify Structure Class.

```
assert ( isstruct ( param ) )
assert ( isa ( param, 'struct' ) )
```

Sets the input parameter *param* to the MATLAB class `struct` (structure). For example, to set the class of input *U* to a `struct`, call:

```
...
assert(isstruct(U));
...
```

or

```
...
assert(isa(U, 'struct'));
...
```

Note If you set the class of an input parameter to `struct`, you must specify the properties of all fields in the order they appear in the structure definition.

Specify Any Size.

```
assert ( all ( size ( param == [dims] ) ) )
```

Sets the input parameter *param* to the size specified by dimensions *dims*. For example, to set the size of input *U* to a 3-by-2 matrix, call:

```
...
```

```
assert(all(size(U)== [3 2]));
...
```

Specify Scalar Size.

```
assert ( isscalar (param) )
assert ( all ( size (param == [ 1 ] ) )
```

Sets the size of input parameter *param* to scalar. For example, to set the size of input U to scalar, call:

```
...
assert(isscalar(U));
...
```

or

```
...
assert(all(size(U)== [1]));
...
```

Specify Real Input.

```
assert ( isreal (param) )
```

Specifies that the input parameter *param* is real. For example, to specify that input U is real, call:

```
...
assert(isreal(U));
...
```

Specify Complex Input.

```
assert ( ~isreal (param) )
```

Specifies that the input parameter *param* is complex. For example, to specify that input U is complex, call:

```
...
assert(~isreal(U));
...
```

Specify numerictype of Fixed-Point Input.

```
assert ( isEqual ( numerictype ( fiparam ), T ) )
```

Sets the numerictype properties of fi input parameter *fiparam* to the numerictype object *T*. For example, to specify the numerictype property of fixed-point input U as a signed numerictype object T with 32-bit word length and 30-bit fraction length, use the following code:

```
%#eml
...
% Define the numerictype object.
T = numerictype(1, 32, 30);

% Set the numerictype property of input U to T.
assert(isEqual(numerictype(U),T));
...
```

Specify fimath of Fixed-Point Input.

```
assert ( isEqual ( fimath ( fiparam ), F ) )
```

Sets the fimath properties of fi input parameter *fiparam* to the fimath object *F*. For example, to specify the fimath property of fixed-point input U so that it saturates on integer overflow, use the following code:

```
%#eml
...
% Define the fimath object.
F = fimath('OverflowMode','saturate');

% Set the fimath property of input U to F.
assert(isEqual(fimath(U),F));
...
```

Note If you do not specify the fimath properties using assert, emlc uses the MATLAB default fimath value.

Specify Multiple Properties of Input.

```
assert ( function1 ( params ) &&
        function2 ( params ) &&
        function3 ( params ) && ... )
```

Specifies the class, size, and complexity of one or more inputs using a single `assert` function call. For example, the following code specifies that input `U` is a double, complex, 3-by-3 matrix, and input `V` is a 16-bit unsigned integer:

```
%#eml
...
assert(isa(U, 'double') &&
      ~isreal(U) &&
      all(size(U) == [3 3]) &&
      isa(V, 'uint16'));
...
...
```

Rules for Using assert Function

Follow these rules when using the `assert` function to specify the properties of primary function inputs:

- Call `assert` functions at the beginning of the primary function, before any control-flow operations such as `if` statements or subroutine calls.
- Do not call `assert` functions inside conditional constructs, such as `if`, `for`, `while`, and `switch` statements.
- Use the `assert` function with Embedded MATLAB Coder only for specifying properties of primary function inputs before converting your MATLAB code to C/C++ code.
- If you set the class of an input parameter to `fi`, you must also set its `numerictype`, see “Specify numerictype of Fixed-Point Input” on page 28-35. You can also set its `fimath` properties, see “Specify fimath of Fixed-Point Input” on page 28-35. If you do not set the `fimath` properties, `emlc` uses the MATLAB default `fimath` value.
- If you set the class of an input parameter to `struct`, you must specify the class, size, and complexity of all fields in the order they appear in the structure definition.

Example: Specifying General Properties of Primary Inputs

In the following code excerpt, a primary MATLAB function `emcspecgram` takes two inputs: `pennywhistle` and `win`. The code specifies the following properties for these inputs:

Input	Property	Value
<code>pennywhistle</code>	class	<code>int16</code>
	size	220500-by-1 vector
	complexity	<code>real</code> (by default)
<code>win</code>	class	<code>double</code> (by default)
	size	1024-by-1 vector
	complexity	<code>real</code> (by default)

```
%#eml
function y = emcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16'));
assert(all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double'));
assert(all(size(win) == [nfft 1]));
...
```

Alternatively, you can combine property specifications for one or more inputs inside `assert` commands, as follows:

```
%#eml
function y = emcspecgram(pennywhistle,win)
nx = 220500;
nfft = 1024;
assert(isa(pennywhistle,'int16') && all(size(pennywhistle) == [nx 1]));
assert(isa(win, 'double') && all(size(win) == [nfft 1]));
...
```

Example: Specifying Properties of Primary Fixed-Point Inputs

In the following example, the primary MATLAB function `emcsqrtfi` takes one fixed-point input: `x`. The code specifies the following properties for this input:

Property	Value
<code>class</code>	<code>fi</code>
<code>numerictype</code>	<code>numerictype</code> object <code>T</code> , as specified in the primary function
<code>fimath</code>	<code>fimath</code> object <code>F</code> , as specified in the primary function
<code>size</code>	<code>scalar</code> (by default)
<code>complexity</code>	<code>real</code> (by default)

```
%#eml
function y = emcsqrtfi(x)
T = numerictype('WordLength',32,'FractionLength',23,
    'Signed',true);
F = fimath('SumMode','SpecifyPrecision',
    'SumWordLength',32,'SumFractionLength',23,
    'ProductMode','SpecifyPrecision',
    'ProductWordLength',32,'ProductFractionLength',23);
assert(isfi(x));
assert(isequal(numerictype(x),T));
assert(isequal(fimath(x),F));

y = sqrt(x);
```

Example: Specifying Class and Size of Scalar Structure

Assume you have defined `S` as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',int8(4));
```

Here is code that specifies the class and size of `S` and its fields when passed as an input to your MATLAB function:

```
%#eml
function y = fcn(S)
```

```
% Specify the class of the input as struct.
assert(isstruct(S));

% Specify the class and size of the fields r and i
% in the order in which you defined them.
assert(isa(S.r,'double'));
assert(isa(S.i,'int8'));

...
```

Note In most cases, Embedded MATLAB Coder uses defaults when you don't explicitly specify values for properties — except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore in the example above, an `assert` function specifies that field `S.r` is of type `double`, even though `double` is the default.

Example: Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume you have defined `S` as the following 2-by-2 array of MATLAB structures:

```
S = struct('r',{double(1), double(2)},'i',{int8(4), int8(5)});
```

The following code specifies the class and size of each field of structure input `S` using the first element of the array:

```
%#eml
function y = fcn(S)

% Specify the class of the input S as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
% based on the first element of the array.
assert(all(size(S) == [2 2]));
assert(isa(S(1).r,'double'));
assert(isa(S(1).i,'int8'));
```

Note In most cases, Embedded MATLAB Coder uses defaults when you don't explicitly specify values for properties — except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore in the example above, an `assert` function specifies that field `S(1).r` is of type `double`, even though `double` is the default.

Choosing Your Target

In this section...

- “Types of Targets” on page 28-41
- “Specifying the Target to the Compiler” on page 28-42
- “Specifying a Language for Embeddable Code Generation” on page 28-43
- “Relationship of Targets and Configuration Objects” on page 28-43
- “Location of Generated Files” on page 28-44
- “Specifying main Functions for C/C++ Executables” on page 28-44

Types of Targets

Embedded MATLAB Coder can generate code for the following targets:

To Generate:	Use Target Reserved Word:
MEX function	<code>mex</code> (the default)
Embeddable C/C++ code and compile it to an executable	<code>rtw:exe</code>
Note When you choose this target, you must provide a C/C++ file that contains the <code>main</code> function for generating the executable, as described in “Specifying main Functions for C/C++ Executables” on page 28-44.	
Embeddable C/C++ code and compile it to a library	<code>rtw:lib</code> or <code>rtw</code>

Specifying the Target to the Compiler

After choosing your target, invoke Embedded MATLAB Coder with the `-T` option and pass the appropriate reserved word as its argument (see the description of `emlc` in the Real-Time Workshop Reference). For example, suppose you have a primary function called `MyFcn`. The following table shows how to specify different targets when compiling `MyFcn`:

To Generate:	Use This Command:
MEX function	<pre>emlc -T mex MyFcn or emlc MyFcn</pre> <p>(translates MATLAB function to MEX function by default)</p>
Embeddable C/C++ code and compile it to an executable	<pre>rtwcfg = emlcoder.RTWConfig; rtwcfg.CustomSource = 'main.c'; rtwcfg.CustomInclude = 'c:\myfiles'; emlc -T rtw:exe -s rtwcfg MyFcn</pre> <p>Note</p> <ul style="list-style-type: none"> <code>emlcoder.RTWConfig</code> is the configuration object for GRT-based targets. Use <code>emlcoder.RTWConfig('ert')</code> for ERT-based targets. You must specify a main function for generating an embeddable C/C++ executable. See “Specifying main Functions for C/C++ Executables” on page 28-44. C is the default language for code generation with Real-Time Workshop. To generate C++ code, see “Specifying a Language for Embeddable Code Generation” on page 28-43
Embeddable C/C++ code and compile it to a library	<pre>emlc -T rtw:lib MyFcn or emlc -T rtw MyFcn</pre>

Specifying a Language for Embeddable Code Generation

`emlc` can generate libraries or executables in C or C++ for embedded applications. C is the default language for the `rtw`, `rtw:exe`, and `rtw:lib` targets (see “Types of Targets” on page 28-41). To specify a language explicitly:

- 1 Select the appropriate compiler for your target language.
- 2 Create a configuration object for embeddable code generation. For example:

```
rtwcfg = emlcoder.RTWConfig;
```

- 3 Set the `TargetLang` property to 'C' or 'C+'. For example:

```
rtwcfg.TargetLang = 'C++';
```

See Also

- To specify the target language using a GUI, see “Modifying Configuration Objects Using Dialog Boxes” on page 28-21.
- For more information about using configuration objects, see “Configuring Your Environment for Code Generation” on page 28-16.
- To select a compiler, see “Setting Up the C/C++ Compiler” on page 28-9.

Relationship of Targets and Configuration Objects

There is a relationship between targets and configuration objects. For example, there is a Real-Time Workshop configuration object, `emlcoder.RTWConfig`, that allows you to define properties for the `rtw:lib` and `rtw:exe` targets, that is, for generating embeddable code as a C/C++ library or executable. The best practice is to match your configuration object with your target when you invoke Embedded MATLAB Coder, as in this example:

```
rtwcfg = emlcoder.RTWConfig;
emlc -T rtw:lib -s rtwcfg myMFn
```

However, if you do not specify a target, but do provide a configuration object, Embedded MATLAB Coder determines the target from the configuration object. For example, consider these sample commands:

```
rtwcfg = emlcoder.RTWConfig;
emlc -s rtwcfg myMFn
```

Normally, when you do not specify a target explicitly, Embedded MATLAB Coder assumes the default target, `mex`, and generates MEX code. However in this case, Embedded MATLAB Coder assumes the desired target is `rtw:lib`, based on the configuration object `rtwcfg`, and therefore, generates embeddable C/C++ code as a library instead of generating MEX code.

For more information about configuration objects, see “Configuring Your Environment for Code Generation” on page 28-16. For more information about invoking Embedded MATLAB Coder, see “Compiling Your MATLAB File” on page 28-46.

Location of Generated Files

By default, Embedded MATLAB Coder generates files in output folders based on your target. See “Generated Files and Locations” on page 28-65.

Specifying main Functions for C/C++ Executables

When you choose `rtw:exe` as your target, you must provide a C/C++ file that contains the `main` function for generating your executable. You can specify the C/C++ file by using the `CustomSource` and `CustomInclude` properties of the Real-Time Workshop configuration object (see “Configuring Your Environment for Code Generation” on page 28-16). The `CustomInclude` property indicates the location of C/C++ files specified by `CustomSource`.

To specify a main function for a C/C++ executable, follow these steps:

- 1 Create a Real-Time Workshop configuration object, as in this command:

```
rtwcfg = emlcoder.RTWConfig;
% Configuration object for GRT-based target
```

- 2 Set the `CustomSource` property to the name of the C/C++ source file that contains the `main` function, as in this command:

```
rtwcfg.CustomSource = 'main.c';
```

- 3** Set the `CustomInclude` property to the location of `main.c`, as in this command:

```
rtwcfg.CustomInclude = 'c:\myfiles';
```

- 4** Generate the C/C++ executable using the appropriate command line options, as in this example:

```
emlc -T rtw:exe -s rtwcfg myMFunction
```

Embedded MATLAB Coder compiles and links the main function with the C/C++ code it generates from `myMFunction.m`.

Compiling Your MATLAB File

In this section...

“Before Compiling Your MATLAB Code” on page 28-46

“Running Embedded MATLAB Coder” on page 28-46

“Specifying Compiler Options” on page 28-47

“Specifying Custom Files to Build” on page 28-47

Before Compiling Your MATLAB Code

Before compiling MATLAB code, make sure you have followed the prerequisite steps in “Workflows for Converting MATLAB Code to C/C++ Code” on page 28-5 or “Workflow for Converting MATLAB Code to a MEX Function” on page 28-6.

Running Embedded MATLAB Coder

You run Embedded MATLAB Coder from the MATLAB command prompt by using the `emlc` function. The basic command is:

```
emlc M_fcn
```

By default, `emlc` performs the following actions:

- Searches for the function *M_fcn* stored in the file *M_fcn.m* as specified in “Compile Path Search Order” on page 28-11.
- Compiles *M_fcn*, checking for compliance with Embedded MATLAB subset.
- If there are no errors or warnings, generates a platform-specific MEX file in the current folder, using the naming conventions described in “Naming Conventions” on page 28-12.
- If there are errors, does not generate a MEX file, but produces an error report in a default output folder, as described in “Generated Files and Locations” on page 28-65. Error reports are described in “Working with Compilation Reports” on page 28-70.
- If there are warnings, but no errors, generates a platform-specific MEX file in the current folder, but does report the warnings.

Specifying Compiler Options

You can modify this default behavior by specifying one or more compiler options with `emlc`, separated by spaces on the command line. For example, to generate a C/C++ code executable you must specify the target `rtw:exe` and the location of a main function, as in this example:

```
emlc -T rtw:exe C:\custom\myMain.c M_fcn
```

For a complete list of compiler options and how to use them, see `emlc` in the Real-Time Workshop Function Reference.

Specifying Custom Files to Build

In addition to your MATLAB file, you can specify the following types of custom files to include in the build for embeddable C/C++ code generation:

File Extension	Description
.c	Custom C file
.cpp	Custom C++ file
.h	Custom header file
.o , .obj	Custom object file
.a , .lib, .so	Library
.tmf	Template makefile for custom Real-Time Workshop buildsSee “Customizing Template Makefiles” on page 21-86.

Compiling More Than One Entry-Point MATLAB Function

In this section...

- “Advantages of Compiling More Than One Entry-Point Function” on page 28-48
- “How to Compile More Than One Entry-Point Function” on page 28-48
- “How to Call an Entry-Point Function in a MEX Function” on page 28-50
- “How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code” on page 28-50

Advantages of Compiling More Than One Entry-Point Function

Compiling more than one entry-point function to generate a single C/C++ library allows you to:

- Create C/C++ libraries containing multiple compiled MATLAB files to integrate with larger C/C++ applications.
- Share code efficiently between library functions.
- Communicate between library functions using shared memory.

Compiling more than one entry-point function to generate a MEX function allows you to validate entry-point interactions in MATLAB before creating a C/C++ library.

How to Compile More Than One Entry-Point Function

To compile more than one entry-point function, use the following syntax, where `global_options` applies to all functions, `fun_1` through `fun_n`, and `options_n` applies only to the preceding function `fun_n`.

```
emlc -global_options fun_1 -options_1 ... fun_n -options_n
```

By default, `emlc`:

- Generates a MEX function in the current folder. `emlc` derives the name of the MEX function from the name of the first entry-point function, `fun_1`.

- Stores generated files in the subfolder `emcprj/mexfcn/fun_1/`.

If you specify an output file name `out_fun` using the `-o` option, `emlc` stores the generated files in the subfolder `emcprj/mexfcn/out_fun/`. For more information on setting compilation options, see “Automatic C MEX Generation Dialog Box for Embedded MATLAB Coder” in the Real-Time Workshop Reference documentation.

Example: Generating a MEX Function with Two Entry-Point Functions Using `emlc`

Generate a MEX function with two entry-point functions, `ep1` and `ep2`. Function `ep1` takes one input, a single scalar, and `ep2` takes two inputs, a double scalar and a double vector. Name the generated MEX function `sharedmex` using the `-o` option.

```
emlc -o sharedmex ep1 -eg single(0) ep2 -eg { 0, zeros(1,1024) }
```

`emlc` generates a MEX function named `sharedmex` in the current folder and stores generated files in the subfolder `emcprj/mexfcn/sharedmex`.

Note By default, `emlc` generates a MEX function named `ep1` in the subfolder `emcprj/mexfcn/ep1`.

Example: Generating a C/C++ Library with Two Entry-Point Functions Using `emlc`

Generate embeddable C/C++ code and compile it to a library for two entry-point functions, `ep1` and `ep2`. Function `ep1` takes one input, a single scalar, and `ep2` takes two inputs, a double scalar and a double vector. Use the `-T rtw:lib` option to specify that the target is a library. Name the generated library function `sharedlib` using the `-o` option.

```
emlc -T rtw:lib -o sharedlib ep1 -eg single(0) ep2 ...
      -eg { 0, zeros(1,1024) }
```

`emlc` generates C/C++ library code in the `emcprj\rtwlib\sharedlib` folder.

Note By default, `emlc` generates a library function named `ep1.lib` in the subfolder `emcprj/rtwlib/ep1`.

For information on viewing entry-point functions in the compilation report, see “Working with Compilation Reports” on page 28-70.

How to Call an Entry-Point Function in a MEX Function

To call an entry-point function in a MEX function that has more than one entry point, use this syntax:

```
MEX_Function('entry_point_function_name',
    ... entry_point_function_param1,
    ... , entry_point_function_paramn)
```

Example: Calling an Entry-Point Function in a MEX Function

Consider a MEX function, `sharedmex`, that has entry-point functions `ep1` and `ep2`. Entry-point function `ep1` takes one single scalar input and `ep2` takes two inputs, a double scalar and a double vector.

To call `ep1` with an input parameter `u`, enter this command:

```
sharedmex(`ep1', u)
```

To call `ep2` with input parameters `u` and `v`, enter this command:

```
sharedmex(`ep2', u, v)
```

How to Call an Entry-Point Function in a C/C++ Library Function from C/C++ Code

To call an entry-point function in a C/C++ library function from C/C++ code, write a `main` function in C/C++ that:

- Includes the generated header files, which contain the function prototypes for the entry-point functions.
- Calls the initialize function before calling the entry-point functions for the first time.

- Calls the terminate function after calling the entry-point functions for the last time.
- Configures your target to integrate this custom C/C++ main function with your generated code, as described in “Integrating Custom C/C++ Code with Generated Code” on page 28-95.
- Generates the C/C++ executable using `emlc` with the `-T rtw:exe` option.

See the example, “Example: Calling a C Library Function from C Code” on page 28-92.

Generating C/C++ Code from MATLAB Code That Uses Global Data

In this section...

- “Workflow Overview” on page 28-52
- “Declaring Global Variables” on page 28-52
- “Defining Global Data” on page 28-53
- “Synchronizing Global Data with MATLAB” on page 28-54
- “Limitations of Using Global Data” on page 28-58

Workflow Overview

To generate C/C++ code from MATLAB code that uses global data:

- 1** Declare the variables as global in your code.
- 2** Define and initialize the global data before using it.

For more information, see “Defining Global Data” on page 28-53.

- 3** Compile your code using `emlc`.

For simulation, if you use global data, you must also specify whether you want to synchronize this data between MATLAB and Embedded MATLAB code. For more information on synchronization, see “Synchronizing Global Data with MATLAB” on page 28-54.

Declaring Global Variables

When using global data, you must first declare the global variables in your Embedded MATLAB code. Consider the `use_globals` function that uses two global variables `AR` and `B`.

```
function y = use_globals(u)
%#eml
% Turn off inlining to make
% generated code easier to read
```

```

eml.inline('never');
% Declare AR and B as global variables
global AR;
global B;
AR(1) = u + B(1);
y = AR * 2;

```

Defining Global Data

You can define global data either in the MATLAB global workspace or at the command line. If you do not initialize global data at the command line, `emlc` looks for the variable in the MATLAB global workspace. If the variable does not exist, `emlc` generates an error.

Defining Global Data in the MATLAB Global Workspace

To compile the `use_globals` function described in “Declaring Global Variables” on page 28-52 using `emlc`:

- 1 Define the global data in the MATLAB workspace. At the MATLAB prompt, enter:

```

global AR B;
AR = ones(4);
B=[1 2 3];

```

- 2 Compile the function to generate a MEX file named `use_globalsx`.

```
emlc -o use_globalsx use_globals
```

Defining Global Data at the Command Line

To define global data at the command line, use the `emlc -global` option. For example, to compile the `use_globals` function described in “Declaring Global Variables” on page 28-52, specify two global inputs `AR` and `B` at the command line.

```
emlc -o use_globalsx -global {'AR',ones(4),'B',[1 2 3]} use_globals
```

Defining Variable-Sized Global Data. To provide initial values for variable-sized global data, use an example object with the `-global` flag. For example, to specify a global variable `g1` that has an initial value `[1 1]` and upper bound `[2 2]`, enter:

```
emlc foo -global {'g1',emlcoder.egs([1 1],[2 2])}
```

For a detailed explanation of `emlcoder.egs` syntax, see “Specifying Variable-Size Inputs” in the *Embedded MATLAB User’s Guide*.

Synchronizing Global Data with MATLAB

Why Synchronize Global Data?

The Embedded MATLAB subset and MATLAB each have their own copies of global data. To ensure consistency, you must synchronize their global data whenever the two interact. If you do not synchronize the data, their global variables might differ. The level of interaction determines when to synchronize global data. For more information, see “When to Synchronize Global Data” on page 28-54.

When to Synchronize Global Data

By default, synchronization between Embedded MATLAB global data and MATLAB occurs at MEX function entry and exit and for all extrinsic calls. This behavior ensures maximum consistency between Embedded MATLAB and MATLAB.

To improve performance, you can:

- Select to synchronize only at MEX function entry and exit points.
- Disable synchronization when the global data does not interact.
- Choose whether to synchronize before and after each extrinsic call.

The following table summarizes which global data synchronization options to use. To learn how to set these options, see “How to Synchronize Global Data” on page 28-56.

Global Data Synchronization Options

If you want to...	Set the global data synchronization mode to:	Synchronize before and after extrinsic calls?
Ensure maximum consistency when all extrinsic calls modify global data.	At MEX-function entry, exit and extrinsic calls (default)	Yes. Default behavior.
Ensure maximum consistency when most extrinsic calls modify global data, but a few do not.	At MEX-function entry, exit and extrinsic calls (default)	Yes. Use the <code>eml.extrinsic -sync:off</code> option to turn off synchronization for the extrinsic calls that do not affect global data.
Ensure maximum consistency when most extrinsic calls do not modify global data, but a few do.	At MEX-function entry and exit	Yes. Use the <code>eml.extrinsic -sync:on</code> option to synchronize only the calls that modify global data.
Maximize performance when synchronizing global data, and none of your extrinsic calls modify global data.	At MEX-function entry and exit	No.
Communicate between Embedded MATLAB files only. No interaction between MATLAB and Embedded MATLAB global data.	Disabled	No.

How to Synchronize Global Data

To control global data synchronization, set the global data synchronization mode and select whether to synchronize extrinsic functions. For guidelines on which options to use, see “When to Synchronize Global Data” on page 28-54.

You can control the global data synchronization mode from the command line or the Automatic C-MEX Generation dialog box. You control the synchronization of data with extrinsic functions using the `eml.extrinsic -sync:on` and `-sync:off` options.

Controlling the Global Data Synchronization Mode from the Command Line.

- 1 Define the compiler options object in the MATLAB workspace by issuing a constructor command:

```
mexcfg = emlcoder.MEXConfig
```

- 2 From the command line, set the `GlobalDataSyncMethod` property to `SyncAtEntryAndExits` or `NoSync`, as applicable. For example:

```
mexcfg.GlobalDataSyncMethod = 'SyncAtEntryAndExits';
```

For more information, see “Automatic C MEX Generation Dialog Box for Embedded MATLAB Coder”.

- 3 Use the `mexcfg` configuration object when compiling your code by specifying it using the `-s` compilation option. For example:

```
emlc -s mexcfg myFile
```

Controlling the Global Data Synchronization Mode Using the Automatic C-MEX Generation Dialog Box.

- 1 Define the MEX configuration object in the MATLAB workspace by issuing a constructor command:

```
mexcfg = emlcoder.MEXConfig;
```

- 2 Open the Automatic C-MEX Generation dialog box:

```
open mexcfg
```

- 3** Set the global data synchronization mode as applicable, and click **Apply**.
- 4** Use the `mexcfg` configuration object when compiling your code by specifying it using the `-s` compilation option. For example:

```
emlc -s mexcfg myFile
```

Controlling Synchronization for Extrinsic Function Calls. To control whether synchronization between MATLAB and Embedded MATLAB global data occurs before and after you call an extrinsic function, use the `eml.extrinsic-sync:on` and `-sync:off` options.

By default, global data is:

- Synchronized before and after each extrinsic call if the global data synchronization mode is `At MEX-function entry, exit and extrinsic calls`. If you are sure that certain extrinsic calls do not affect global data, turn off synchronization for these calls using the `-sync:off` option. For example, if functions `foo1` and `foo2` do not affect global data, turn off synchronization for these functions:

```
eml.extrinsic('-sync:off', 'foo1', 'foo2');
```

- Not synchronized if the global data synchronization mode is `At MEX-function entry and exit`. If the code has a few extrinsic calls that affect global data, turn on synchronization for these calls using the `-sync:on` option. For example, if functions `foo1` and `foo2` do affect global data, turn on synchronization for these functions:

```
eml.extrinsic('-sync:on', 'foo1', 'foo2');
```

- Not synchronized if the global data synchronization mode is `Disabled`. When synchronization is disabled, you cannot control the synchronization for specific extrinsic calls. The `-sync:on` option has no effect.

Limitations of Using Global Data

You cannot use global data with the `eml.cstructname` function. This function does not support global variables.

How emlc Generates Code

In this section...

- “Partitioning Generated Files” on page 28-59
- “How to Select the File Partitioning Method” on page 28-59
- “Partitioning Generated Files When You Generate One C/C++ File Per MATLAB File” on page 28-60
- “Generated Files and Locations” on page 28-65
- “File Partitioning and Inlining” on page 28-67

Partitioning Generated Files

By default, during code generation with Real-Time Workshop software, the `emlc` function partitions the code to match your MATLAB file structure. This one-to-one mapping lets you easily correlate your files generated in C/C++ with the compiled MATLAB code. `emlc` cannot produce the same one-to-one correspondence for MATLAB functions that are inlined in generated code (see “File Partitioning and Inlining” on page 28-67).

Alternatively, you can select to generate all C/C++ functions into a single file. For more information, see “How to Select the File Partitioning Method” on page 28-59. This option facilitates integrating your Embedded MATLAB code with existing embedded software.

How to Select the File Partitioning Method

For MEX Targets

Use the MEX configuration object `FilePartitionMethod` option. For example, to compile the file `foo.m` and generate one C/C++ file for each MATLAB function:

- 1 Create a MEX configuration object and set the `FilePartitionMethod` option.

```
mexcfg = emlcoder.MEXConfig;
mexcfg.FilePartitionMethod = 'MapMFileToCFile';
```

2 Pass the configuration object to `emlc` using the `-s` option:

```
emlc foo -s mexcfg -O disable:inline  
% Disable inlining to ensure one C/C++ file for each MATLAB function
```

Alternatively, you can modify code generation parameters using dialog boxes. To learn how to work with these dialog boxes, see “Configuration Parameters for Embedded MATLAB Coder” in the Real-Time Workshop Reference documentation.

For C/C++ Executable Targets

Use the Real-Time Workshop configuration object `FilePartitionMethod` option. For example, to generate all C/C++ functions into a single file:

1 Create a Real-Time Workshop configuration object and set the `FilePartitionMethod` option.

```
rtwcfg = emlcoder.RTWConfig;  
rtwcfg.FilePartitionMethod = 'SingleFile';
```

2 Pass the configuration object to `emlc` using the `-s` option:

```
emlc foo -s mexcfg -T rtw:lib
```

Alternatively, you can modify code generation parameters using dialog boxes. To learn how to work with these dialog boxes, see “Configuration Parameters for Embedded MATLAB Coder” in the Real-Time Workshop Reference documentation.

Partitioning Generated Files When You Generate One C/C++ File Per MATLAB File

By default, for MATLAB functions that are not inlined, `emlc` generates one C/C++ file for each MATLAB file. In this case, `emlc` partitions generated C/C++ code so that it corresponds to your MATLAB files.

How `emlc` Partitions Top-Level MATLAB Functions

For each top-level MATLAB function, `emlc` generates one C/C++ source, header, and object file with the same name as the MATLAB file.

For example, suppose you define a simple function **foo** that calls the function **identity**. The source file **foo.m** contains the following code:

```
function y = foo(u,v) %#eml
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);
```

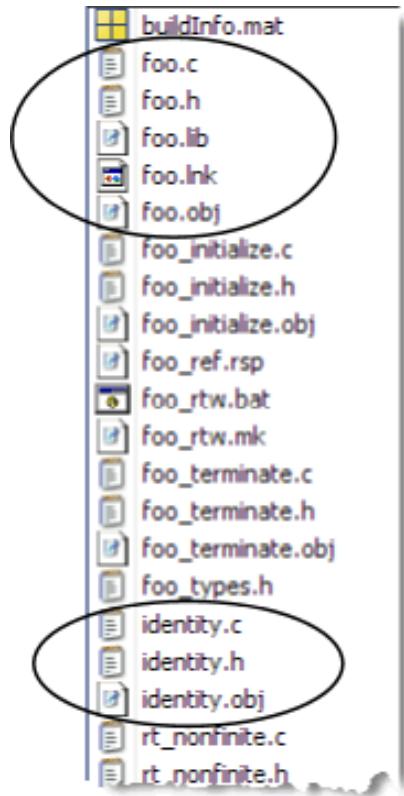
Here is the code for **identity.m**:

```
function y = identity(u) %#eml
y = u;
```

To generate a C library for **foo.m**, execute the following command at the MATLAB prompt:

```
emlc -T rtw:lib -O disable:inline foo
```

emlc generates source, header, and object files for **foo** and **identity** in your output folder:



How emlc Partitions Subfunctions

For each subfunction, emlc generates code in the same C/C++ file as the calling function. For example, suppose you define a function `foo` that calls a subfunction `identity`:

```

function y = foo(u,v) %#eml
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);

function y = identity(u)
y = u;

```

To generate a C++ library, make sure you select a C++ compiler and execute the following commands:

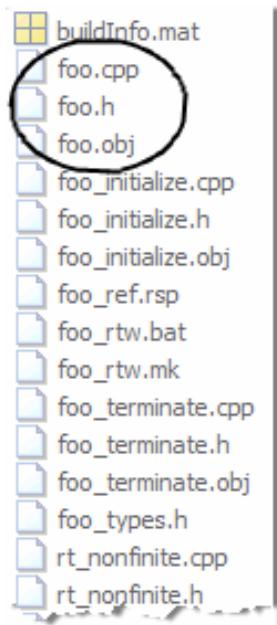
- 1** Select C++ as your target language:

```
rtwcfg = emlcoder.RTWConfig
rtwcfg.TargetLang='C++'
```

- 2** Generate the C++ library:

```
emlc -T rtw:lib -s rtwcfg foo
```

emlc inlines the code for the **identity** subfunction in the primary function **foo**:



Here is an excerpt of the generated code in **foo.cpp**:

```
...
/* Function Definitions */
real_T foo(real_T eml_u, real_T eml_v)
{
    return (real_T)(real32_T)eml_u + eml_v;
}
...
```

How emlc Partitions Overloaded Functions

An overloaded function is a function that has multiple implementations to accommodate different classes of input. For each implementation (that is not inlined), `emlc` generates a separate C/C++ file with a unique numeric suffix.

For example, suppose you define a simple function `multiply_defined`:

```
%#eml
function y = multiply_defined(u)

y = u+1;
```

You then add two more implementations of `multiply_defined`, one to handle inputs of type `single` (in an `@single` subfolder) and another for inputs of type `double` (in an `@double` subfolder).

To call each implementation, define the function `call_multiply_defined`:

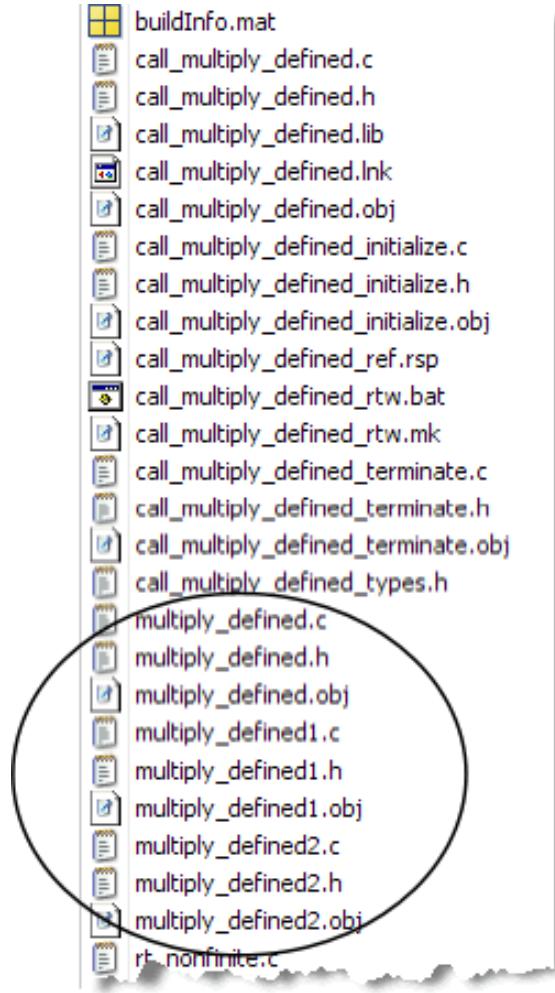
```
%#eml
function [y1,y2,y3] = call_multiply_defined

y1 = multiply_defined(int32(2));
y2 = multiply_defined(2);
y3 = multiply_defined(single(2));
```

Next, execute the following command to generate C code for the overloaded function `multiply_defined`:

```
emlc -T rtw:lib -O disable:inline call_multiply_defined
```

`emlc` generates C source, header, and object files for each implementation of `multiply_defined`, as highlighted. Numeric suffixes ensure unique file names.



For more information about overloaded functions, see “Overloaded MATLAB Functions” in the MATLAB Programming Fundamentals documentation.

Generated Files and Locations

The types and locations of generated files depend on the target that you specify. For all targets, `emlc` generates HTML reports if errors or warnings occur during compilation or if you explicitly request a report.

Generated Files for MEX Targets

By default, `emlc` generates the following files for MEX function (`mex`) targets:

Type of Files	Location
Platform-specific MEX files	Current folder
MEX, and C/C++ source, header, and object files	<code>emcprj/mexfcn/function_name</code>
HTML reports	<code>emcprj/mexfcn/function_name/html</code>

Generated Files for C/C++ Executable Targets

By default, `emlc` generates the following files for embeddable C/C++ executable (`rtw:exe`) targets:

Type of Files	Location
C/C++ source, header, and object files	<code>emcprj/rtwexe/function_name</code>
HTML reports	<code>emcprj/rtwexe/function_name/html</code>

Generated Files for C/C++ Library Targets

By default, `emlc` generates the following files for embeddable C/C++ library (`rtw:lib` or `rtw`) targets:

Type of Files	Location
C/C++ source, library, header, and object files	<code>emcprj/rtwlib/function_name</code>
HTML reports	<code>emcprj/rtwlib/function_name/html</code>

Changing Names and Locations of Generated Files

You can change the name and location of generated files by using the compiler options `-o` and `-d` (see `emlc` in the Real-Time Workshop Function Reference).

File Partitioning and Inlining

How `emlc` partitions generated C/C++ code depends on whether you choose to generate one C/C++ file for each MATLAB file and whether you inline your MATLAB functions.

If you...	<code>emlc</code> ...
Generate all C/C++ functions into a single file and disable inlining	Generates a single C/C++ file without inlining any functions.
Generate all C/C++ functions into a single file and enable inlining	Generates a single C/C++ file. Inlines functions whose sizes fall within the inlining threshold.
Generate one C/C++ file for each MATLAB file and disable inlining	Partitions generated C/C++ code to match MATLAB file structure. See “Partitioning Generated Files When You Generate One C/C++ File Per MATLAB File” on page 28-60.
Generate one C/C++ file for each MATLAB file and enable inlining	Places inlined functions in the same C/C++ file as the function into which they are inlined. Even when you enable inlining, <code>emlc</code> does not inline all functions, only those whose sizes fall within the inlining threshold. For MATLAB functions that are not inlined, <code>emlc</code> partitions the generated C/C++ code as described above.

Tradeoffs Between File Partitioning and Inlining

Weighing file partitioning against inlining represents a trade-off between readability, efficiency, and ease of integrating your Embedded MATLAB code with existing embedded software.

If You Generate...	Generated C/C++ Code	Advantages	Disadvantages
All C/C++ functions into a single file	Does not match MATLAB file structure	Easier to integrate with existing embedded software	Difficult to map C/C++ code to original MATLAB file
One C/C++-file for each MATLAB file and enable inlining	Does not exactly match MATLAB file structure	Program executes faster	Difficult to map C/C++ code to original MATLAB file
One C/C++-file for each MATLAB file and disable inlining	Matches MATLAB file structure	Easy to map C/C++ code to original MATLAB file	Program runs less efficiently

Disabling Inlining to Ensure File Partitioning

Inlining is enabled by default. Therefore, to ensure that `emlc` generates one C/C++ file for each top-level MATLAB function, you must:

- Choose to generate one C/C++ file for each top-level MATLAB function. For more information, see “How to Select the File Partitioning Method” on page 28-59.
- Explicitly disable inlining, either globally or for individual MATLAB functions.

How to Disable Inlining Globally. To disable inlining of functions, use the `-O disable:inline` option with `emlc`. For example:

```
emlc foo -T RTW:LIB -O disable:inline
```

For more information, see the description of `emlc` in the Real-Time Workshop Reference documentation.

How to Disable Inlining for Individual Functions. To disable inlining for an individual MATLAB function, add the directive `eml.inline('never')`; on a separate line in the source MATLAB file after the function signature, as in this example:

```
function y = foo(u,v) %#eml
eml.inline( never );
s = single(u);
d = double(v);
y = double(identity(s)) + identity(d);
```

The directive applies only to the function in which it appears. In this example, inlining is disabled for function `foo`, but not for `identity`, a top-level function defined in a separate MATLAB file and called by `foo`. To disable inlining for `identity`, add this directive after its function signature in the source file `identity.m`. For more information, see `eml.inline` in the Embedded MATLAB Reference documentation.

For a more efficient way to disable inlining for both functions, use the `-O` compiler option described in “How to Disable Inlining Globally” on page 28-68.

Correlating C/C++ Code with Inlined Functions

To correlate the C/C++ that you generate with the original inlined functions, add comments in the MATLAB code to identify the function. These comments will appear in the C/C++ code and help you map the generated code back to the original MATLAB functions.

Modifying the Inlining Threshold

To change inlining behavior, you might want to adjust the inlining threshold, a compiler options parameter. To modify this parameter, see “Compiler Options Dialog Box” in the Embedded MATLAB documentation.

Working with Compilation Reports

In this section...

- “About Compilation Reports” on page 28-70
- “Generating Compilation Reports” on page 28-70
- “Names and Locations of Compilation Reports” on page 28-71
- “Opening Compilation Reports” on page 28-72
- “Description of Compilation Reports” on page 28-73
- “Viewing Your MATLAB Code” on page 28-74
- “Viewing Call Stack Information” on page 28-75
- “Viewing the Generated C/C++ Code” on page 28-77
- “Viewing the Compilation Summary Information” on page 28-77
- “Viewing Error and Warning Messages” on page 28-77
- “Viewing Variables in Your MATLAB Code” on page 28-79
- “Viewing Target Build Information” on page 28-85
- “Keyboard Shortcuts for the Compilation Report” on page 28-85
- “Embedded MATLAB Compilation Report Limitations” on page 28-86

About Compilation Reports

Embedded MATLAB Coder produces reports in HTML format at compile time to help you debug your MATLAB code and verify compliance with the Embedded MATLAB subset.

Generating Compilation Reports

Embedded MATLAB Coder automatically produces a compilation report if it detects errors or warnings.

You can also use the option to request reports even when there are no errors or warnings.

The report provides links to your MATLAB code and C/C++ code files. It also provides compile-time type information for the variables and expressions in your MATLAB code. This information simplifies finding sources of error messages and aids understanding of type propagation rules.

For example, to generate an embeddable C/C++ library for the file `demo.m` with a compilation report using `emlc`, at the MATLAB command line, enter:

```
emlc -T rtw:lib -report demo
```

Names and Locations of Compilation Reports

Embedded MATLAB Coder produces compilation reports in the following locations. The top-level html file at each location is `index.html`.

- For MEX targets (`mex`):

output_folder/mexfcn/primary_function_name/html

- For C/C++ executable targets (`rtw:exe`):

output_folder/rtwexe/primary_function_name/html

- For C/C++ library targets (`rtw:lib`):

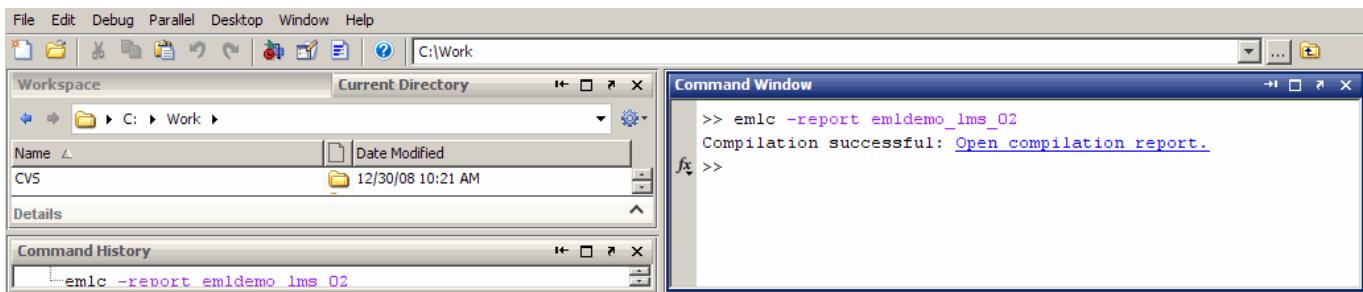
output_folder/rtwlblib/primary_function_name/html

Note The default output folder is `emcprj`, but you can specify a different folder with the `-d` option (see `emlc` in the Real-Time Workshop reference documentation).

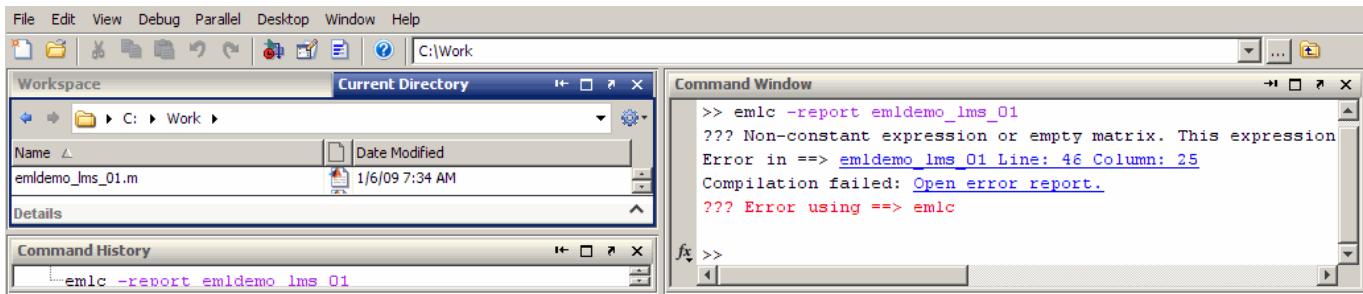
Opening Compilation Reports

If you specify the `-launchreport` option, the compilation report opens automatically.

To open the compilation report if no compilation errors occurred, click the **Open compilation report** link in the MATLAB Command Window.



To open the error report if compilation errors occurred, click the **Open error report** link in the MATLAB Command Window.



Description of Compilation Reports

When you compile MATLAB files with the `-report` option, Embedded MATLAB Coder generates an HTML report. The following example shows a report for a successful compilation.

The screenshot shows the MATLAB Coder compilation report interface. At the top, there are tabs for "MATLAB code", "Call stack", and "C code". The "Function" dropdown is set to "emldemo_lms_02". The "Callers" dropdown is set to "Select a function call-site". On the left, there is a filter panel with options like "Filter function list by attributes", "Filter by: Size", and "Filter: 1 x 1". Below the filter panel, it says "Filter matches: 1/1 functions" and highlights "emldemo_lms_02" with a blue selection bar. The main area displays the MATLAB code for the function:

```

43 % Adaptation step size:
44 mu = 4/(32*1024);
45
46 % Signal length:
47 N = length(distorted);
48
49 if length(desired) == N
50
51     % Filter coefficients:
52     weights = zeros(L,1);
53
54     % Zero Pad Input Signal:
55     distorted_pad = [zeros(L-1,1); distorted];
56
57     % Pre-allocate output and error signals:
58     signal_out = zeros(N,1);
59     err = zeros(N,1);
60
61     % Filter Update Loop:
62     % ...

```

At the bottom of the code editor, there is a toolbar with tabs: "Summary", "All Messages (0)", "Variables", and "Target Build Log". The "Summary" tab is selected. The summary information is as follows:

- C source code generated on: 21-Oct-2009 15:00:55
- Coding target: RTW Library
- Number of errors: 0

The report provides the following information, as applicable:

- MATLAB code information, including a list of all functions and their compilation status
- Call stack information, providing information on the nesting of function calls
- Links to generated C/C++ code files
- Summary of compilation results, including type of target and number of warnings or errors
- List of all error and warning messages

- List of all variables in your MATLAB code
- Target build log that records compilation and linking activities

Viewing Your MATLAB Code

To view your MATLAB code, click on the **MATLAB code** tab. The compilation report displays the code for the function highlighted in the list on this tab.

The **MATLAB code** tab provides:

- A list of the MATLAB functions that have been compiled. The report displays icons next to each function name to indicate whether compilation was successful:
 - Errors in function.
 - Warnings in function.
 - Successful compilation, no errors or warnings.
- A filter control, **Filter function list by attributes**, that you can use to sort your functions by:
 - Size
 - Complexity
 - Class

Viewing Subfunctions

The compilation report annotates the subfunction with the name of the parent function in the list of functions on the **MATLAB code** tab.

For example, if the MATLAB function `fcn1` contains the subfunction `subfcn1` and `fcn2` contains the subfunction `subfcn2`, the report displays:

```
fcn1 > subfcn1
fcn2 > subfcn2
```

Viewing Specializations

If your Embedded MATLAB function calls the same function with different types of inputs, the compilation report numbers each of these **specializations** in the list of functions on the **MATLAB code** tab.

For example, if the function `fcn` calls the function `subfcn` with different types of inputs:

```
function y = fcn(u) %#eml
% Specializations
y = y + subfcn(single(u));
y = y + subfcn(double(u));
```

The compilation report numbers the specializations in the list of functions.

```
fcn > subfcn > 1
fcn > subfcn > 2
```

Viewing Call Stack Information

The compilation report provides call stack information:

- On the **Call stack** tab.
- In the list of **Callers**.

If a function is called from more than one function, this list provides details of each call-site. Otherwise, the list is disabled.

Viewing Call Stack Information on the Call stack Tab

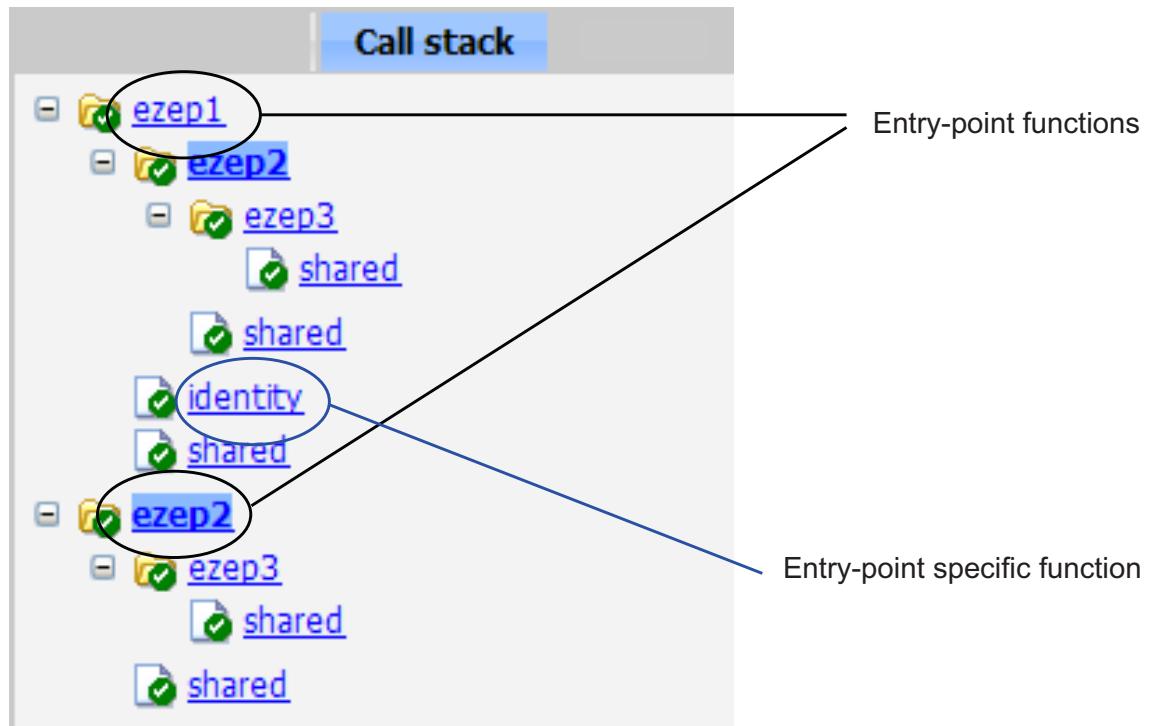
To view call stack information, click the **Call stack** tab.

The call stack lists the functions in the order that the top-level function calls them. It also lists the subfunctions that each function calls.

If there is more than one entry-point function, the call stack displays a separate tree for each entry point. You can distinguish easily between shared and entry-point specific functions. If you click a shared function, the report

highlights all instances of this function. If you click an entry-point specific function, the report highlights only that instance.

For example, in the following call stack, `ezep1` and `ezep2` are entry-point functions, and `identity` is an entry-point specific function, called only by `ezep1`. Functions `ezep3` and `shared` are shared functions.



Viewing Function Call-Sites in the Callers List

If a function is called from more than one function, this list provides details of each call site. To navigate between call-sites, select a call-site from the **Callers** list. If the function is not called more than once, this list is disabled.

For example, if you select the entry-point function `ezep2` in the call stack, the **Callers** list displays the other call-site in `ezep1`.

The screenshot shows a MATLAB compilation report interface. At the top, it says "Function: [ezep2](#)". Below that is the code listing:

```

1 function y = ezep2(u) %#eml
2 y = shared\(ezep3\(u\)\);

```

To the right, there are two dropdown menus under "Callers":

- "Select a function call-site:" (with a dropdown arrow)
- A dropdown menu showing "Select a function call-site:" followed by "ezep1 at 2".

Viewing the Generated C/C++ Code

To view a list of the generated C/C++ files, click the **C-code** tab. The compilation report displays a list of the generated files. Click a file in the list to view the code in the code pane.

Navigating to C/C++ Code Source Files

When viewing C/C++ code in the code pane, click the blue link to the source file at the top of the pane to open the associated source code file in the MATLAB editor.

Viewing Type Definitions

The compilation report provides links to the definitions of data types. When viewing C/C++ code in the code pane, click the blue link for a data type to see its definition.

Viewing the Compilation Summary Information

To view a summary of the compilation results, including type of target and number of errors or warnings, click the **Summary** tab.

Viewing Error and Warning Messages

Embedded MATLAB Coder automatically reports errors and warnings. If errors occur during compilation, Embedded MATLAB Coder does not generate code. The report lists the messages in the order that the compiler detects them. It is best practice to address the first message in the list, because often subsequent errors and warnings are related to the first message. If compilation produces warnings, but no errors, Embedded MATLAB Coder does generate code.

The compilation report provides information about errors and warnings by:

- Listing all errors and warnings in the **All Messages** tab. The report lists these messages in chronological order.
- Highlighting all errors and warnings in the **MATLAB code** pane

Viewing Errors and Warnings in the All Messages Tab

If errors or warnings occurred during compilation, click the **All Messages** tab to view a complete list of these messages. The compilation report marks messages as follows:



Error



Warning

To locate the offending line of code for an error or warning in the list, click the message in the list. The compilation report highlights errors in the list and MATLAB code in red and warnings in orange. Click the blue line number next to the offending line of code in the MATLAB code pane to go to the error in the source file.

Note You can only fix errors in the source file.

Click the blue line number next to the line of code with an error to go to the offending code in the source file

The report underlines errors in red

The screenshot shows a MATLAB code editor window with the following code:

```

Function: emldemo_lms_01_mlint
Callers: Select a function call-site: □

56     for n = 1:N
57         signal_out(n) = weights' * distorted_pad(n:n+L-1);
58         err(n) = desired(n) - signal_out(n) ;
59         weights = weights + mu*err(n)*distorted_pad(n:n+L-1);
60     end
61
62     % Plot Impulse Response:
63     figure;
64     stem(weights);
65     title(['IMLS Filter -- Version #' num2str(k) ...
66             ' -- Impulse Response']);
67
68 else
69
70     error('Lengths of input signals are not equal');
71
72 end
73
74 end
75

```

The line `figure;` is highlighted in red, indicating an error. Below the code, a table lists two errors:

Summary	All Messages (2)	Variables		
Order	Type	Function	Line	Description
1	✖	emldemo_lms_01_mlint	63	Function 'figure' implicitly resolved in the MATLAB workspace.
2	✖	emldemo_lms_01_mlint	64	The function 'stem' is not supported by Embedded MATLAB

An arrow points from the text "The report highlights the selected error in the list" to the second row of the table, which corresponds to the `stem` function error.

The report highlights the selected error in the list

Viewing Error and Warning Information in Your MATLAB Code

If errors or warnings occurred during compilation, the compilation report underlines them in your MATLAB code. The report underlines errors in red and warnings in orange. To learn more about a particular error or warning, place your pointer over the underlined text.

Viewing Variables in Your MATLAB Code

The compilation report provides compile-time type information for the variables and expressions in your MATLAB code, including name, type, size, complexity, and class. The report also provides type information for fixed-point data types including word length and fraction length. You can use

this type information to find sources of error messages and to understand type propagation rules.

You can view information about the variables in your MATLAB code by:

- Viewing the list in the **Variables** tab
- Placing your pointer over the variable name in your MATLAB code

Viewing Variables in the Variables Tab

To view a list of all the variables in your MATLAB function, click the **Variables** tab. The compilation report displays a complete list of variables in the order they appear in the function selected in the **MATLAB code** tab. Clicking a variable in the list highlights all instances of that variable, and scrolls the MATLAB code pane so that the first instance is in view.

Tip You can sort the variables by clicking the column headings in the **Variables** tab. To sort the variables by multiple columns, hold down the **Shift** key when clicking the column headings.

The report provides the following information about each variable, as applicable. The report only displays a column if at least one variable in the code has information in that column. For example, if the code does not contain any fixed-point data types, the report does not display the **DT mode**, **WL** or **FL** columns.

- Name
- Type
- Size
- Complexity
- Class
- **DataTypeMode** (DT mode) — for fixed-point data types only. For more information, see “**DataTypeMode**” in the Fixed-Point Toolbox documentation.

- Signed — sign information for built-in data types, signedness information for fixed-point data types
- Word length (WL) — for fixed-point data types only
- Fraction length (FL) — for fixed-point data types only

Note For more information on viewing fixed-point data types, see “Working with Fixed-Point Compilation Reports” in the Fixed-Point Toolbox documentation.

Viewing Information about Variable-Size Arrays in the Variables Tab.

For variable-size arrays, the size field includes information on the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon :.

Variable	Type	Size	Complex	Class
out	Output variable	:5x:5	No	double
out2	Output variable	:4x2	No	double
in	Input variable	1x1	No	double
in2	Input variable	:4x:4	No	double
t1	Local variable	:3x:3	No	double

The size of the variable `in` is fixed at '1 x 1'.

The array `in2` is variable-sized. Its maximum computed size is '4 x 4'.

If Embedded MATLAB technology cannot compute the maximum size of a variable-sized array, the compilation report displays the size as :?.

Summary		All Messages (8)	Variables	
Order	Type	Function	Location	Description
1	×	my_emldemo_lms_01_mlint	Line 0047	Computed maximum size is not bounded. Static memory allocation requires all sizes to be bounded. The computed size is [:? x 1].
2	×	my_emldemo_lms_01_mlint	Line 0050	Computed maximum size is not bounded. Static memory allocation requires all sizes to be bounded. The computed size is [:? x 1].
3	×	my_emldemo_lms_01_mlint	Line 0058	Computed maximum size is not bounded. Static memory allocation requires all sizes to be bounded. The computed size is [:? x 1].

'?:' denotes that the computed maximum size is not bounded



For more information on how to use the size information for variable-sized arrays, see “Working with Variable-Size Data” in the Embedded MATLAB documentation.

Viewing Renamed Variables in the Variables Tab. If your Embedded MATLAB function reuses a variable with different size, type, or complexity, whenever possible, Embedded MATLAB creates separate uniquely named variables in the generated code. (For more information, see “Reusing the Same Variable with Different Properties” in the Embedded MATLAB documentation.) The compilation report numbers the renamed variables in the list in the **Variables** tab. When you place your pointer over a renamed variable, the compilation report highlights only the instances of this variable that share the same data type, size, and complexity.

For example, suppose your code uses the variable `t` in a for-loop to hold a scalar double, and reuses it outside the for-loop to hold a 5×5 matrix. The compilation report displays two variables, `t>1` and `t>2` in the list in the **Variables** tab.

```

5 if all(all(u))
6     % First time t is used to hold a scalar double value
7     t = mean(mean(u)) / numel(u);
8     u = u - t;
9 end
10 % t is reused to hold a matrix of doubles
11 t = u;

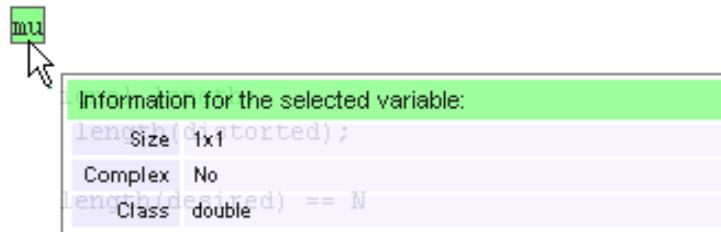
```

Summary	All Messages (0)	Variables		
Variable	Type	Size	Complex	Class
u	Input variable	5 x 5	No	double
t > 1	Local variable	5 x 5	No	double
t > 2	Local variable	1 x 1	No	double

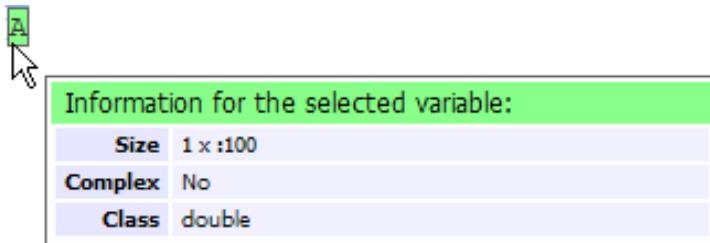
Viewing Information about Variables and Expressions in Your Embedded MATLAB Function Code

To view information about a particular variable or expression in your Embedded MATLAB function code, place your pointer over the variable name or expression in the MATLAB code pane. The compilation report highlights variables and expressions in different colors:

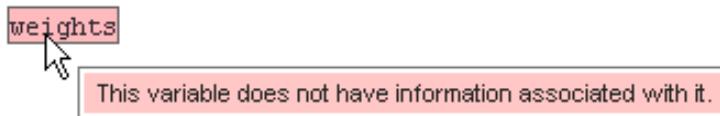
Green, when the variable has data type information at this location in the code.



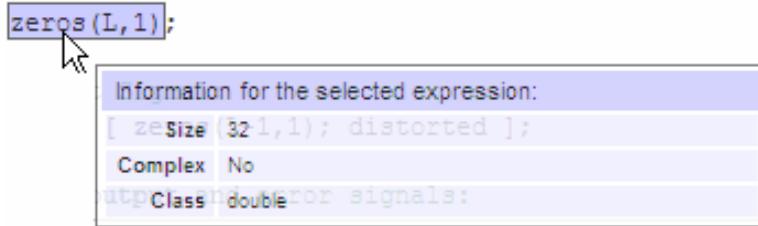
For variable-sized arrays, the size field includes information on the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon :. Here the array A is variable-sized with a maximum computed size of 1 x 100.



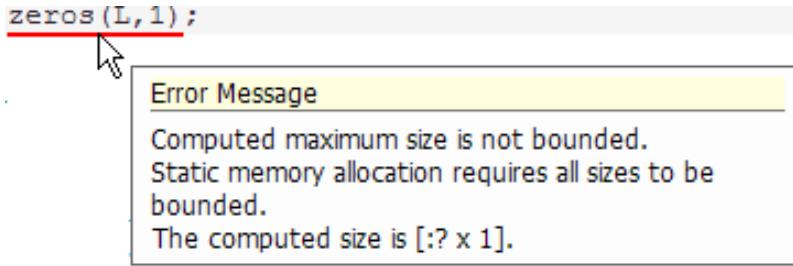
Pink, when the variable has no data type information.



Purple, information about expressions. You can also view information about expressions in your MATLAB code. Place your pointer over an expression in the MATLAB code pane. The compilation report highlights expressions in purple and provides more detailed information.



Red, when there is error information for an expression. If the Embedded MATLAB software cannot compute the maximum size of a variable-sized array, the compilation report underlines the variable name and provides error information.



Viewing Target Build Information

If Embedded MATLAB Coder compiles your MATLAB code without errors, it provides target build information in the **Target Build Log** tab, including:

- Build folder

Clicking this link changes the MATLAB current folder to the build folder.

- Make wrapper

The name of the batch file that Embedded MATLAB Coder used for this build.

- Build log

Summary	All Messages (12)	Variables	Target Build Log
Build Parameters			
Build directory	C:\Work\emcp\mexfcn\warn		
Make wrapper	warn_mex.bat		
Build Log			
<pre> 1 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 2 warn_data.c 3 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 4 warn.c 5 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 6 warn_initialize.c 7 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 8 warn_terminate.c 9 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 10 morphfcn.c 11 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 12 warn_api.c 13 cl /c /Zp8 /GR /W3 /EHs /D_CRT_SECURE_NO_DEPRECATED /D_SCL_SECURE_NO_DEPRECATED /D_SECURE_SCL=0 /DMATLAB_MEX_FILE 14 warn_mex.c 15 link /dll /export:mexFunction /LIBPATH:"\\\\\\mathworks\\\\devel\\\\JOBARC~1\\\\Aslrtw\\\\SNAPSHT\\\\2009_0~1\\\\current\\\\ma 16 Creating library warn.x and object warn.exp 17 mt -outputresource:"warn.mexw32;2" -manifest "warn.mexw32.manifest" 18 Microsoft (R) Manifest Tool version 5.2.3790.2014 19 Copyright (c) Microsoft Corporation 2005. 20 All rights reserved. </pre>			

Keyboard Shortcuts for the Compilation Report

You can use the following keyboard shortcuts to navigate between the different panes in the compilation report. Once you have selected a pane, use the Tab key to advance through data in that pane.

To select ...	Use...
MATLAB Code Tab	Ctrl+m
Call Stack Tab	Ctrl+k
C Code Tab	Ctrl+c
Code Pane	Ctrl+w
Summary Tab	Ctrl+s
All Messages Tab	Ctrl+a
Variables Tab	Ctrl+v
Target Build Log Tab	Ctrl+t

Embedded MATLAB Compilation Report Limitations

The compilation report displays information about the variables and expressions in your Embedded MATLAB code with the following limitations:

varargin and varargout

The report does not support varargin and varargout arrays.

Loop Unrolling

The report does not display the correct information for unrolled loops.

Dead Code

The report does not display information about any dead code.

Structures

The report does not provide complete information about structures.

- You cannot expand a **struct** in the list on the **Variables** tab to view all its fields
- In the **MATLAB code** pane, the report does not provide information about all structure fields in the **struct()** constructor

- In the **MATLAB code** pane, if a structure has a nonscalar field, and an expression accesses an element of this field, the report does not provide information for the field

Column Headings on Variables Tab

If you scroll down through the list of variables, the report does not display the column headings on the **Variables** tab.

Multiline Matrices

In the **MATLAB code** pane, the report does not support selection of multiline matrices. It only supports selection of individual lines at a time. For example, if you place your pointer over the following matrix, you cannot select the entire matrix.

```
out1 = [1 2 3;
        4 5 6];
```

The report does support selection of single line matrices.

```
out1 = [1 2 3; 4 5 6];
```

Calling Generated C/C++ Functions

In this section...

- “Conventions for Calling Functions in Generated Code” on page 28-88
- “Calling C/C++ Functions from the Embedded MATLAB Subset” on page 28-88
- “Calling Initialize and Terminate Functions” on page 28-89
- “Calling C/C++ Functions with Multiple Outputs” on page 28-94
- “Calling C/C++ Functions that Return Arrays” on page 28-94

Conventions for Calling Functions in Generated Code

When generating code, Embedded MATLAB Coder (`emlc`) uses the following calling conventions:

- Passes arrays by reference as inputs and returns arrays by reference as outputs.
- Passes scalars by value as inputs unless you optimize your code by using the same variable as both input and output; in that case, `emlc` passes the scalar by reference.
- Returns scalars by value for single-output functions.
- Returns scalars by reference:
 - For functions with multiple outputs.
 - When you use the same variable as both input and output.

For more information about optimizing your code by using the same variable as both input and output, see “Eliminating Redundant Copies of Function Inputs” in the Embedded MATLAB documentation.

Calling C/C++ Functions from the Embedded MATLAB Subset

You can call the C/C++ functions generated for `rtw:lib` targets as custom C/C++ code from MATLAB functions that comply with the Embedded

MATLAB subset. In this scenario, you must use the Embedded MATLAB interface `eml.ceval` to wrap the function calls, as in this example:

```
%#eml
function y = callmyCFunction
y = 1.5;
y = eml.ceval('myCFunction',y);
```

Here, the Embedded MATLAB function `callmyCFunction` calls the custom C/C++ function `myCFunction`, which takes one input argument. For more information about `eml.ceval`, see “Calling C/C++ Functions from the Embedded MATLAB Subset” in the Embedded MATLAB documentation.

Note `eml.ceval` is part of the Embedded MATLAB language, but is not supported in the native MATLAB language.

There are additional requirements for calling C/C++ functions from the Embedded MATLAB subset in the following situations:

- You want to call generated C/C++ libraries (`rtw:lib` targets) or executables (`rtw:exe` targets) from an Embedded MATLAB function. In this case, you must call housekeeping functions generated by Embedded MATLAB Coder, as described in “Calling Initialize and Terminate Functions” on page 28-89.
- You want to call C/C++ functions that are generated from MATLAB functions that have more than one output, as described in “Calling C/C++ Functions with Multiple Outputs” on page 28-94.
- You want to call C/C++ functions that are generated from MATLAB functions that return arrays, as described in “Calling C/C++ Functions that Return Arrays” on page 28-94.

Calling Initialize and Terminate Functions

When you convert a MATLAB function to a C/C++ library function for `rtw:lib` targets or a C/C++ executable for `rtw:exe` targets, Embedded MATLAB Coder automatically generates two housekeeping functions that you must call along with the C/C++ function, as follows:

Housekeeping Function	When to Call
<i>primary_function_name_initialize</i>	Before you call your C/C++ executable or library function for the first time
<i>primary_function_name_terminate</i>	After you call your C/C++ executable or library function for the last time

Note From C/C++ code, you can call these functions directly. However, to call them from MATLAB code that conforms to the Embedded MATLAB subset, you must use the `eml.ceval` interface, as described in “Calling C/C++ Functions from the Embedded MATLAB Subset” in the Embedded MATLAB documentation. `eml.ceval` is part of the Embedded MATLAB subset, but not supported by the native MATLAB language. Therefore, if your MATLAB code uses this interface, you must compile your MATLAB code on the Embedded MATLAB path (see “Adding Folders to Search Paths” on page 28-12).

Example: Calling a C/C++ Library Function from MATLAB Code

This example shows how to call a C/C++ library function from MATLAB code that conforms to the Embedded MATLAB subset.

Suppose you have an MATLAB file `absval.m` that contains the following function:

```
%#eml
function y = absval(u)
y = abs(u);
```

To generate a C library function and call it from MATLAB code, follow these steps:

- 1 Generate the C library for `absval.m` by invoking Embedded MATLAB Coder, as follows:

```
emlc -T rtw:lib absval -eg {0.0}
```

Note Here are key points about this command:

- The `-T rtw:lib` option instructs Embedded MATLAB Coder to generate `absval` as a C library function.
The default target language is C. To change the target language to C++, see “Specifying a Language for Embeddable Code Generation” on page 28-43.
 - Embedded MATLAB Coder creates the library `absval.lib` (or `absval.a` on Linus Torvalds’ Linux) and header file `absval.h` in the folder `/emcprj/rtwlib/absval`. It also generates the functions `absval_initialize` and `absval_terminate` in the C library.
 - The `-eg` option specifies the class, size, and complexity of the primary function input `u` by example, as described in “Defining Input Properties by Example at the Command Line” on page 28-28.
-

2 Write a MATLAB function to call the generated library, as follows:

```
%#eml
function y = callabsval

% Call the initialize function before
% calling the C function for the first time
eml.ceval('absval_initialize');

y = -2.75;
y = eml.ceval('absval',y);

% Call the terminate function after
% calling the C function for the last time
eml.ceval('absval_terminate');
```

Note The MATLAB function `callabsval` uses the interface `eml.ceval` to call the generated C functions `absval_initialize`, `absval`, and `absval_terminate`. You must use this interface to call C functions from the Embedded MATLAB language subset, as described in “Calling C/C++ Functions from the Embedded MATLAB Subset” in the Embedded MATLAB documentation.

- 3** Convert the code in `callabsval.m` to a MEX function so you can call the C library function `absval` directly from the MATLAB prompt. Follow these steps:

- a** Generate the MEX function by invoking Embedded MATLAB Coder as follows:

On Microsoft Windows platforms, use this command:

```
emlc -T mex callabsval emcprj/rtwlib/absval/absval.lib  
emcprj/rtwlib/absval/absval.h
```

On the Linus Torvalds' Linux platform, use this command:

```
emlc -T mex callabsval emcprj/rtwlib/absval/absval.a  
emcprj/rtwlib/absval/absval.h
```

- b** Call the C library by running the MEX function from the MATLAB prompt, like this:

```
callabsval
```

Example: Calling a C Library Function from C Code

This example shows how to call a generated C library function from C code. It uses the C library function `absval` described in “Example: Calling a C/C++ Library Function from MATLAB Code” on page 28-90.

To call a C library function from a C file, follow these steps:

- 1** Write a `main` function in C that does the following:

- Includes the generated header file, which contains the function prototypes for the library function
- Calls the initialize function before calling the library function for the first time
- Calls the terminate function after calling the library function for the last time

Here is an example of a C `main` function that can be used to call the library function `absval`:

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "absval.h"

int main(int argc, char *argv[])
{
    absval_initialize();

    printf("absval(-2.75)=%g\n", absval(-2.75));

    absval_terminate();

    return 0;
}
```

- 2 Configure your target to integrate this custom C main function with your generated code, as described in “Integrating Custom C/C++ Code with Generated Code” on page 28-95.

For example, you can define a Real-Time Workshop configuration object that points to the custom C code by following these steps:

- Create a Real-Time Workshop configuration object, by entering a command like this at the MATLAB prompt:

```
rtwcfg = emlcoder.RTWConfig;
```

- b** Set custom code properties on the configuration object, as in these example command lines:

```
rtwcfg.CustomSource = 'main.c';
rtwcfg.CustomInclude = 'c:\myfiles';
```

- 3** Generate the C executable, by entering a command like this at the MATLAB prompt:

```
emlc -T rtw:exe -s rtwcfg absval;
```

- 4** Call the executable, as in this example command:

```
absval(-2.75);
```

Calling C/C++ Functions with Multiple Outputs

Although Embedded MATLAB Coder can generate C/C++ code from MATLAB functions that have multiple outputs, the generated C/C++ code cannot return multiple outputs directly because the C/C++ language does not support multiple return values. Instead, you can achieve the effect of returning multiple outputs from your C/C++ function by using the `eml.wref` construct with `eml.ceval`, as described in “Returning Multiple Values from C Functions” in the Embedded MATLAB documentation.

Calling C/C++ Functions that Return Arrays

Although Embedded MATLAB Coder can generate C/C++ code from MATLAB functions that return values as arrays, the generated code cannot return arrays *by value* because the C/C++ language is limited to returning single, scalar values. Instead, you can return arrays from your C/C++ function *by reference* as pointers by using the `eml.wref` construct with `eml.ceval`, as described in “Calling C/C++ Functions from the Embedded MATLAB Subset” in the Embedded MATLAB documentation.

Integrating Custom C/C++ Code with Generated Code

In this section...

“About Custom C/C++ Code Integration with Embedded MATLAB Coder” on page 28-95

“Specifying Custom C/C++ Files on the Command Line” on page 28-95

“Specifying Custom C/C++ Files with Configuration Objects” on page 28-95

About Custom C/C++ Code Integration with Embedded MATLAB Coder

You integrate custom C/C++ code with generated C/C++ code by specifying the locations of your external source files, header files, and libraries to Embedded MATLAB Coder. You can specify custom C/C++ files on the command line or with configuration objects.

Specifying Custom C/C++ Files on the Command Line

When you compile MATLAB function with Embedded MATLAB Coder, you can specify custom C/C++ files — such as source, header, and library files — on the command line along with your MATLAB file. For example, suppose you want to generate an embeddable C code executable that integrates a custom C function `myCfcn` with a MATLAB function `myMfcn`. The custom source and header files for `myCfcn` reside in the folder `C:\custom`. You can use the following command to generate the code:

```
emlc -T rtw:exe C:\custom\myCfcn.c C:\custom\myCfcn.h myMfcn
```

Specifying Custom C/C++ Files with Configuration Objects

You can specify custom C/C++ files by setting custom code properties on Real-Time Workshop configuration objects (as described in “Real-Time Workshop Dialog Box Overview” in the Real-Time Workshop Reference documentation). Follow these steps:

- 1 Define a Real-Time Workshop configuration object, as described in “Creating Configuration Objects” on page 28-19.

For example:

```
cc = emlcoder.RTWConfig;
```

- 2 Set one or more of the custom code properties as needed:

Custom Code Property	Description
CustomInclude	<p>Specifies a list of folders that contain custom header, source, object, or library files.</p> <p>Note If your folder path name contains spaces, you must enclose it in double quotes, as in this example:</p> <pre>cc.CustomInclude = '"C:\Program Files\MATLAB\work"'</pre>
CustomSource	Specifies additional custom C/C++ files to be compiled with the MATLAB file.
CustomLibrary	Specifies the names of object or library files to be linked with the generated code.
CustomSourceCode	Specifies code to insert at the top of each generated C/C++ source file.
CustomHeaderCode	Specifies custom code to insert at the top of each generated C/C++ header file.

For example:

```
cc.CustomInclude = 'C:\custom\src C:\custom\lib';
cc.CustomSource = 'cfunction.c';
cc.CustomLibrary = 'chelper.obj library.lib';
cc.CustomSourceCode = '#include "cfunction.h"';
```

- 3 Compile the MATLAB code with the Real-Time Workshop configuration object.

For example:

```
emlc -T rtw:lib -s cc myFunc
```

4 Call custom C/C++ functions as follows:

From:	Call:
C/C++ source code	Custom C/C++ functions directly
MATLAB code, compiled on the Embedded MATLAB path	Custom C/C++ functions using <code>eml.ceval</code> , as described in “Calling C/C++ Functions from the Embedded MATLAB Subset” in the Embedded MATLAB documentation.

For example (from MATLAB code):

```
...
y = 2.5;
y = eml.ceval('myFunc',y);
...
```

For more information, see “Configuring Your Environment for Code Generation” on page 28-16.

Speeding Up Simulation with the Basic Linear Algebra Subprograms (BLAS) Library

In this section...

- “How Embedded MATLAB Functions Use the BLAS Library” on page 28-98
- “When to Disable BLAS Library Support” on page 28-98
- “How to Disable BLAS Library Support” on page 28-99
- “Supported Compilers” on page 28-99

How Embedded MATLAB Functions Use the BLAS Library

The Basic Linear Algebra Subprograms (BLAS) Library is a library of external linear algebra routines optimized for fast computation of low-level matrix operations. By default, Embedded MATLAB functions call BLAS library routines to speed up simulation, except in these cases:

- Your C/C++ compiler does not support the BLAS library
See “Supported Compilers for Generating MEX Functions” on page 28-9.
- The size of the matrix is below a minimum threshold
The Embedded MATLAB subset uses a heuristic to evaluate matrix size against the overhead of calling an external library.
- When you are generating C/C++ code for Embedded MATLAB functions using Real-Time Workshop software.
Embedded MATLAB uses BLAS only for simulation.

When to Disable BLAS Library Support

Consider disabling BLAS library support for Embedded MATLAB functions when:

- You are executing code on a 64-bit platform and the number of elements in a matrix exceeds 32-bits.

In this case, Embedded MATLAB automatically truncates the matrix size to 32 bits.

- Your platform does not provide a robust implementation of BLAS routines.

How to Disable BLAS Library Support

Embedded MATLAB functions enable BLAS library support by default, but you can disable this feature explicitly from the command line or the Compiler Options dialog box. Follow these steps:

- 1 Define the MEX configuration object in the MATLAB workspace by issuing a constructor command, like this:

```
mexcfg = emlcoder.MEXCfg
```

- 2 Disable the BLAS option, either from the command line or Automatic MEX Generation dialog box.

From the command line set the `enableBLAS` property false by issuing a command like this:

```
mexcfg.enableBLAS = false;
```

From the Automatic MEX Generation dialog box, follow these steps:

- a Open the Automatic MEX Generation dialog box by issuing a command like this:

```
open mexcfg
```

- b Uncheck the **Use BLAS library if possible** check box and click **Apply**.

Supported Compilers

The Embedded MATLAB subset uses the BLAS library on all C/C++ compilers **except**:

- Watcom
- Intel
- Borland

The default MATLAB compiler, `lcc`, supports the BLAS library. To install a different C/C++ compiler, use the `mex -setup` command, as described in “Building MEX-Files” in the MATLAB External Interfaces documentation.

Controlling Runtime Checks

In this section...

“Types of Runtime Checks” on page 28-101

“When to Disable Runtime Checks” on page 28-102

“How to Disable Runtime Checks” on page 28-102

Types of Runtime Checks

In simulation, the code generated for your Embedded MATLAB functions includes the following runtime checks and external calls to MATLAB functions.

- Memory integrity checks

These checks detect violations of memory integrity in code generated for Embedded MATLAB functions and stop execution with a diagnostic message.

Caution For safety, these checks are enabled by default. Without memory integrity checks, violations will result in unpredictable behavior.

- Responsiveness checks in code generated for Embedded MATLAB functions

These checks enable periodic checks for Ctrl+C breaks in code generated for Embedded MATLAB functions. Enabling responsiveness checks also enables graphics refreshing.

Caution For safety, these checks are enabled by default. Without these checks the only way to end a long-running execution might be to terminate MATLAB.

- Extrinsic calls to MATLAB functions

Extrinsic calls to MATLAB functions, for example to display results, are enabled by default for debugging purposes. For more information about extrinsic functions, see “Declaring MATLAB Functions as Extrinsic Functions”.

When to Disable Runtime Checks

Generally, generating code with runtime checks enabled results in more generated code and slower simulation than generating code with the checks disabled. Similarly, extrinsic calls are time consuming and have an adverse affect on performance. Disabling runtime checks and extrinsic calls usually results in streamlined generated code and faster simulation, with these caveats:

Consider disabling...	Only if...
Memory integrity checks	You are sure that your code is safe and that all array bounds and dimension checking is unnecessary.
Responsiveness checks	You are sure that you will not need to stop execution of your application using Ctrl+C.
Extrinsic calls	You are only using extrinsic calls to functions that do not affect application results.

How to Disable Runtime Checks

You can disable explicitly from the command line or the Automatic C MEX Generation dialog box.

Disabling Runtime Checks From the Command Line

- 1 Define the MEX configuration object in the MATLAB workspace by issuing a constructor command:

```
mexcfg = emlcoder.MEXConfig
```

- 2** From the command line set the `IntegrityChecks`, `ExtrinsicCalls`, or `ResponsivenessChecks` properties false, as applicable:

```
mexcfg.IntegrityChecks = false;  
mexcfg.ExtrinsicCalls = false;  
mexcfg.ResponsivenessChecks = false;
```

Disabling Runtime Checks Using the Automatic C MEX Generation Dialog Box

- 1** Define the MEX configuration object in the MATLAB workspace by issuing a constructor command:

```
mexcfg = emlcoder.MEXConfig
```

- 2** Open the Automatic C MEX Generation dialog box:

```
open mexcfg
```

- 3** Clear the **Ensure memory integrity**, **Ensure responsiveness**, or **Extrinsic calls** check boxes, as applicable, and click **Apply**.

Debugging Runtime Errors

In this section...

“Viewing Errors in the Runtime Stack” on page 28-104

“Handling Runtime Errors” on page 28-105

Viewing Errors in the Runtime Stack

About the Runtime Stack

The runtime stack is enabled by default for Embedded MATLAB functions. If you encounter runtime errors in your Embedded MATLAB functions, the runtime stack appears automatically in the MATLAB command window. Use the error message and the following stack information to learn more about the source of the error:

- The name of the function that generated the error
- The line number of the attempted operation
- The sequence of function calls that led up to the execution of the function and the line at which each of these function calls occurred

Example Runtime Stack Trace. This example shows the runtime stack trace for function `mlstack`:

```

>> mlstackx(-1)
??? Array bounds error accessing element -1 of x(). The valid index range is 1-4.

Error in ==> mlstack>mayfail at 31      ← Information about where the error occurred
y = x(u);

Error in ==> mlstack>subfcn1 at 5      ← Sequence of function calls leading up to error
switch (mayfail(u))

Error in ==> mlstack at 2
y = subfcn1(u);

```

When to Use the Runtime Stack

The runtime stack is useful during debugging to help you find the source of runtime errors. However, when the stack is enabled, the generated code contains push and pop stack calls, which might slow performance. Consider disabling the runtime stack for faster performance.

How to Disable the Runtime Stack. You can disable the runtime stack by disabling the memory integrity checks as described in “How to Disable Runtime Checks” on page 28-102.

Caution Before disabling the memory integrity checks, you should be sure that your code is safe and that all array bounds and dimension checking is unnecessary.

Handling Runtime Errors

The Embedded MATLAB software propagates error ID's. If you throw an error or warning in your Embedded MATLAB code, you can use the `try-catch` statement to examine the error information and attempt to recover, or clean up and abort. For more information, see “The try-catch Statement” in the *MATLAB Programming Fundamentals* documentation.

Integrating External Code With Generated C and C++ Code

- Chapter 29, “About External Code Integration”
- Chapter 30, “Integrating External Code Using Model Configuration Parameters”
- Chapter 31, “Integrating External Code Using Custom Code Blocks”
- Chapter 32, “Integrating External Code Using S-Functions”



About External Code Integration

- “Integration Options” on page 29-2
- “Reusing Algorithmic Components in Generated Code” on page 29-5
- “Deploying Algorithm Code Within a Target Environment” on page 29-16
- “Generating Target Optimizations Within Algorithm Code” on page 29-21
- “Exporting Generated Algorithm Code for Embedded Applications” on page 29-23
- “Exporting Algorithm Executables for System Simulation” on page 29-26
- “Making External Code Language Compatible With Generated Code” on page 29-27

Integration Options

In this section...

[“Introduction” on page 29-2](#)

[“Types of External Code Integration” on page 29-2](#)

Introduction

The Real-Time Workshop 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 Real-Time Workshop software. *Custom code* is legacy code or any other user-specified lines of code that must be included in the Real-Time Workshop build process. Collectively, legacy and custom code are called *external code*.

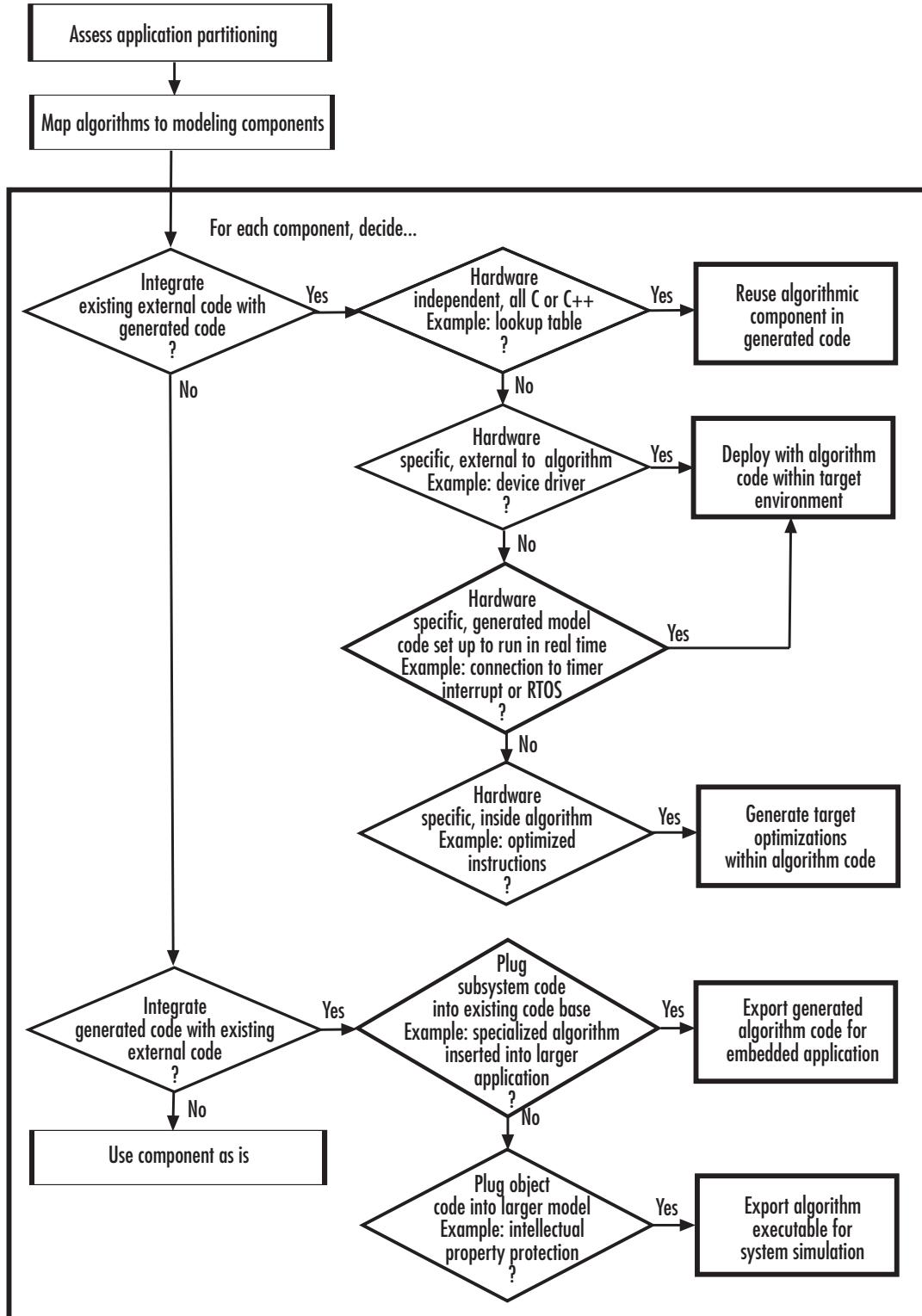
There are two ways that you can achieve external code integration. You can import existing external code into code generated by Real-Time Workshop technology or you can export generated code into an existing external code base. For example, you might want to use generated code as a plug-in function.

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.



Reusing Algorithmic Components in Generated Code

In this section...

- “Examples of Reusable Algorithmic Components” on page 29-5
- “Integrating External MATLAB Code” on page 29-6
- “Integrating External C or C++ Code” on page 29-9
- “Integrating Fortran Code” on page 29-13
- “Other Integration Considerations for Reusable Algorithmic Components” on page 29-13

Examples of Reusable Algorithmic Components

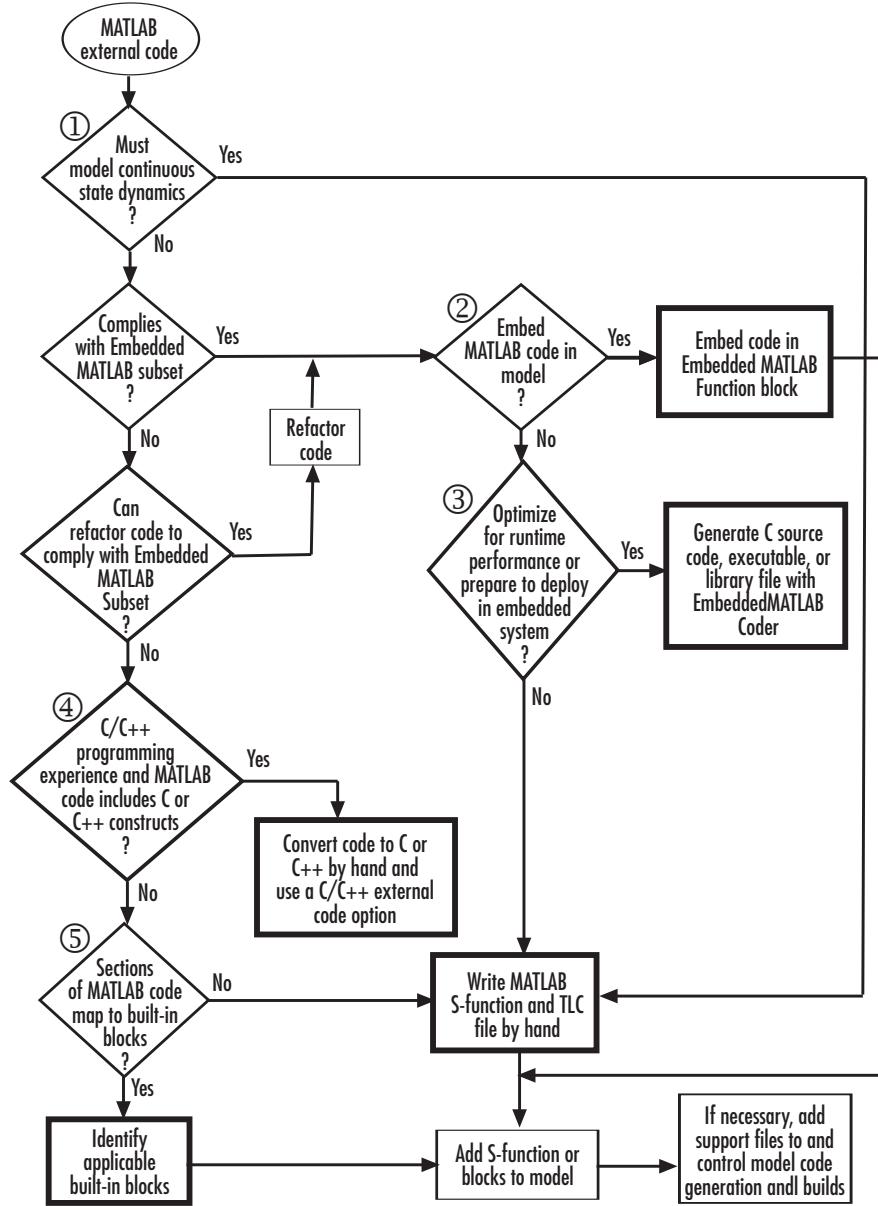
You have several options for integrating reusable algorithmic components with code generated by Real-Time Workshop technology. Such components are hardware-independent and can be verified with Simulink simulation. Examples of such components include:

- Lookup tables
- Math utilities
- Digital filters
- Special integrators
- Proportional–integral–derivative (PID) control modules

Some integration options 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 that 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.

Integrating 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.

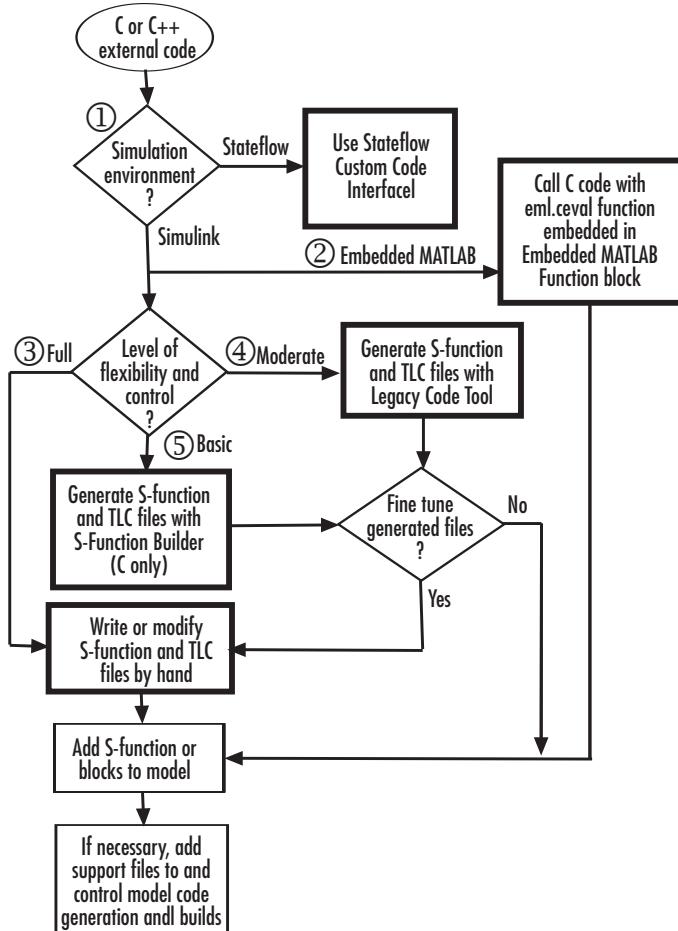


	If...	Then...	For More Information, See...
1	The algorithm must model continuous state dynamics	Write a MATLAB S-function and, for generating code, a corresponding TLC file for the algorithm and add the S-function to your model	<ul style="list-style-type: none"> “Level-2 M-File S-Function Examples” and “Writing S-Functions in M” in the Simulink documentation “Inlining MATLAB File S-Functions”
2	You want to embed MATLAB code directly in the model	Add an Embedded MATLAB Function block to the model and embed the MATLAB code in that block	<ul style="list-style-type: none"> “Using the Embedded MATLAB Function Block” in the Simulink documentation Embedded MATLAB Function block description and “Using the Embedded MATLAB Function Block” in the Simulink documentation
3	You need to optimize runtime performance or prepare to deploy the code in an embedded system	Use Embedded MATLAB Coder to generate a C source code, executable, or library file	<ul style="list-style-type: none"> <code>rtdemo_emlcbasicdemo</code> Chapter 28, “Converting MATLAB Code to C/C++ Code”
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	“Integrating External C or C++ Code” on page 29-9
5	Sections of the external MATLAB code map to built-in blocks	Develop the algorithm in the context of a model, using the applicable built in blocks	<ul style="list-style-type: none"> “Modeling Dynamic Systems” and “Managing Blocks” in the Simulink documentation Chapter 4, “Block Support Considerations”

To embed external MATLAB code in an Embedded MATLAB Function block or generate C or C++ code from MATLAB code with the Embedded MATLAB Coder, the MATLAB code must comply with the Embedded MATLAB subset. For information on how to refactor MATLAB code to comply with the subset, see “Working with the Embedded MATLAB Subset” in the Embedded MATLAB documentation.

Integrating External C or C++ Code

The following diagram identifies common characteristics or requirements for reusable algorithmic components written in C or C++ code and recommends solutions in each case. The table that follows the diagram provides more detail. Collectively, the diagram and table will help you choose the best solution for your application and find more related information.



	If...	Then...	For More Information, See...
1	The external code will simulate in a Stateflow environment	Use the Stateflow custom code interface	<ul style="list-style-type: none"> • <code>sf_custom</code> • “Calling Custom C Code Functions” in the Stateflow documentation
2	Performance is not an issue and you want to quickly embed a call to	Call the C or C++ code with the <code>eml.ceval</code> function from an	<ul style="list-style-type: none"> • <code>eml.ceval</code> function description and “Calling C/C++ Functions from the

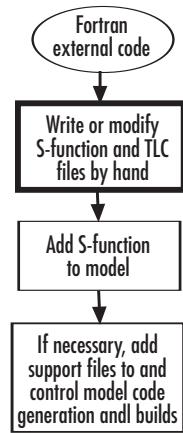
	If...	Then...	For More Information, See...
	external C or C++ code in a model	Embedded MATLAB Function block	<p>“Embedded MATLAB Subset” in the Embedded MATLAB documentation</p> <ul style="list-style-type: none"> “Using the Embedded MATLAB Function Block” in the Simulink documentation Embedded MATLAB Function block description and “Using the Embedded MATLAB Function Block” in the Simulink documentation
3	You want maximum flexibility and the 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	Write an S-function and TLC file manually	<ul style="list-style-type: none"> “C S-Function Examples” and “C++ S-Function Examples” in the Simulink documentation <i>Developing S-Functions</i> in the Simulink documentation Chapter 32, “Integrating External Code Using S-Functions”
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 needed to add the code to a model; optimizing generated code is essential	Use the Legacy Code Tool to generate the necessary S-function and TLC files; optionally, you can fine-tune the generated files manually to better meet application needs	<ul style="list-style-type: none"> <code>rtwdemo_lct_lut_script</code> “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation “Automating Generation of Files for Fully Inlined S-Functions Using Legacy Code Tool” on page 32-25

	If...	Then...	For More Information, See...
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 necessary S-function and TLC files; optionally, you can fine tune the generated files manually to better meet application needs	“Building S-Functions Automatically” in the Simulink documentation

If you must control how Real-Time Workshop 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 a Real-Time Workshop Embedded Coder license. For information on CSCs see the examples `rtwdemo_cscpredef`, `rtwdemo_importstruct`, and `rtwdemo_advsc` and “Creating and Using Custom Storage Classes” in the Real-Time Workshop Embedded Coder documentation.

Integrating Fortran Code

The following diagram shows that to integrate external Fortran code as reusable algorithmic components you must integrate the code by writing and S-function and corresponding TLC file.



For information on how to do this, see “Fortran S-Function Examples” and “Creating Fortran S-Functions” in the Simulink documentation.

Other Integration Considerations for Reusable Algorithmic Components

Note Solutions marked with *EC only* require a Real-Time Workshop Embedded Coder license.

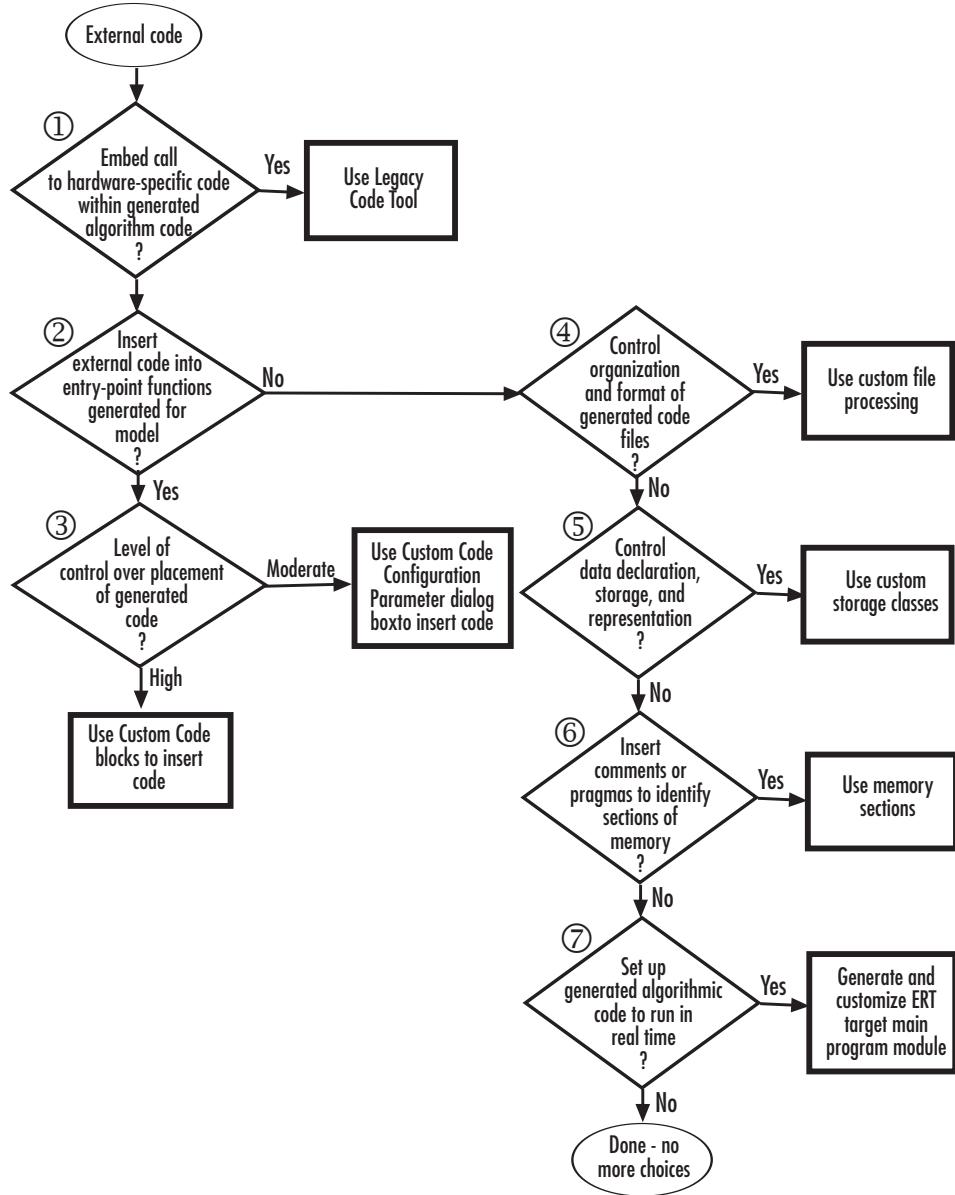
If...	Consider...	For More Information, See...
Generated code must use math functions and operators that are consistent with imported external code	<i>EC only</i> —Using the target function library (TFL) API and Viewer to create, examine, validate, and register function and operator replacement tables	<ul style="list-style-type: none"> <code>rtdemo_tfl_script</code> “Replacing Math Functions and Operators Using Target Function Libraries” in the Real-Time Workshop Embedded Coder documentation
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	<ul style="list-style-type: none"> “Managing Data” in the Simulink documentation “Defining Data Representation and Storage for Code Generation” “Defining Data Representation and Storage for Code Generation” in the Real-Time Workshop Embedded Coder documentation
Style and format of identifiers in generated code must be consistent with style and format applied in imported external code	In the Configuration Parameters dialog box, on the Symbols pane, configure identifier naming for the generated code	<ul style="list-style-type: none"> <code>rtdemo_symbols</code> <code>rtdemo_namerules</code> “Configuring Generated Identifiers” on page 20-3 and “Configuring the Appearance of Generated Identifiers” in the Real-Time Workshop Embedded Coder

If...	Consider...	For More Information, See...
Use of comments in generated code must match the use of comments in imported external code	In the Configuration Parameters dialog box, on the Comments pane, configure comments for the generated code	<ul style="list-style-type: none"> • <code>rtdemo_comments</code> • “Configuring Code Comments” on page 20-2 and “Customizing Comments in Generated Code” in the Real-Time Workshop Embedded Coder
Style of code, such as style and usage of parentheses, must be match the style used in imported external code	<i>EC only</i> —In the Configuration Parameters dialog box, on the Code Style pane, configure the style for the generated code	<ul style="list-style-type: none"> • <code>rtdemo_parentheses</code> • “Controlling Code Style” in the Real-Time Workshop Embedded Coder

Deploying Algorithm Code Within a Target Environment

Real-Time Workshop 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 a Real-Time Workshop Embedded Coder license.

	If You Need To...	Then...	For More Information, See...
1	Embed a call to hardware-specific code, such as a device driver, within generated algorithm code	Use the Legacy Code Tool	<ul style="list-style-type: none"> “Integrating Device Drivers” on page 21-142 “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation “Automating Generation of Files for Fully Inlined S-Functions Using Legacy Code Tool” on page 32-25
2	Insert target-specific C or C++ code into entry-point functions that Real-Time Workshop 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	<ul style="list-style-type: none"> <code>rtwdemo_slcustcode</code> Chapter 31, “Integrating External Code Using 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 Real-Time Workshop > Interface pane of the Configuration Parameters dialog box	<ul style="list-style-type: none"> Integrating the Generated Code into the External Environment Chapter 30, “Integrating External Code Using Model Configuration Parameters”

	If You Need To...	Then...	For More Information, See...
4	Control the organization and format of code files generated for a model—for example, placement of code in sections, inclusion of banners, calls to generated entry-point functions, generation of a main program module	<i>EC only</i> —Use the custom file processing components—code generation template (CGT) files, code template API, and custom file processing (CFP) templates (the API and CFP templates require TLC programming knowledge)	<ul style="list-style-type: none"> • <code>rtdemo_codetemplate</code> • “Configuring Templates for Customizing Code Organization and Format” in the Real-Time Workshop Embedded Coder documentation
5	Control how the Real-Time Workshop product declares, stores, and represents signals, tunable parameters, block states, and data objects in generated code	<i>EC only</i> — Design (create) and apply custom storage classes	<ul style="list-style-type: none"> • <code>rtdemo_cscpredef</code> • <code>rtdemo_importstruct</code> • <code>rtdemo_advsc</code> • “Creating and Using Custom Storage Classes” in the Real-Time Workshop 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	<i>EC only</i> — Use the memory section capability	<ul style="list-style-type: none"> • <code>rtdemo_memsec</code> • “Inserting Comments and Pragmas in Generated Code” in the Real-Time Workshop Embedded Coder documentation
7	Set up generated algorithmic code to run in real time—within the context of a real-time operating system (RTOS) or on hardware that is not running an operating system (bare board)	<i>EC only</i> —Generate and customize an ERT target main (harness) program module (<code>ert_main.c</code> or <code>ert_main.cpp</code>) for the model	<ul style="list-style-type: none"> • Chapter 34, “Interfacing With a Real-Time Operating System” • “Setting Up Generated Code To Interface With Components in the Run-Time Environment” in the Real-Time Workshop Embedded Coder documentation

	If You Need To...	Then...	For More Information, See...
			<ul style="list-style-type: none">• “Interfacing With Hardware That is Not Running an Operating System (Bare Board)”

Generating 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 a Real-Time Workshop Embedded Coder license.

If...	Then...	For More Information, See
You want to optimize target performance (speed and memory) of model code by replacing default math functions and operators with target-specific code	<i>EC only</i> —Implement function and operator replacements by using the target function library (TFL) API and Viewer to create, examine, validate, and register hardware-specific replacement tables	<ul style="list-style-type: none"> • <code>rtdemo_tfl_script</code> • “Replacing Math Functions and Operators Using Target Function Libraries” in the Real-Time Workshop Embedded Coder documentation
You want to control how Real-Time Workshop technology declares, stores, and represents signals, tunable parameters, block states, and data objects in generated code	<i>EC only</i> —Design (create) and apply custom storage classes	<ul style="list-style-type: none"> • <code>rtdemo_cscpredef</code> • <code>rtdemo_importstruct</code> • <code>rtdemo_advsc</code> • “Creating and Using Custom Storage Classes” in the Real-Time Workshop Embedded Coder documentation

Note To simulate an algorithm that includes target-specific elements in a host environment, you must create code equivalent that is equivalent to the target code and can run in the host environment.

Exporting 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 Real-Time Workshop technology.

Note Solutions marked with *EC only* require a Real-Time Workshop Embedded Coder license.

If You Need To...	Then...	For More Information, See...
Insert C or C++ code into specific entry-point functions that Real-Time Workshop generates for interfacing with the external code	Use Custom Code blocks	<ul style="list-style-type: none"> <code>rtdemo_slcustcode</code> Chapter 31, “Integrating External Code Using Custom Code Blocks”
Pass composite data	Represent the data in the model as a vector or bus	<ul style="list-style-type: none"> <code>rtdemo_scalarrep</code> <code>rtdemo_sibus</code> “Using Composite Signals” in the Simulink documentation “Optimizing Code Generated for Vector Assignments” on page 26-47 and “Optimizing Buses for Code Generation” in the Real-Time Workshop Embedded Coder documentation

If You Need To...	Then...	For More Information, See...
Read from or write to a specific region or area of memory	<i>EC only</i> — Set up a Data Store Memory block in the model to represent the area of memory and define the area with the built-in advanced custom storage class (CSC) GetSet	<ul style="list-style-type: none"> “GetSet Custom Storage Class Example” “GetSet Custom Storage Class for Data Store Memory” in the Real-Time Workshop Embedded Coder documentation
Generate a C++ class interface—encapsulated model data (properties) and model entry-point functions (methods)—to the model code	<i>EC only</i> — Configure and generate the C++ encapsulation interface in the Configuration Parameters dialog box or programmatically in the MATLAB command window or with a script	<ul style="list-style-type: none"> “C++ Encapsulation Quick-Start Example” “Controlling Generation of Encapsulated C++ Model Interfaces” in the Real-Time Workshop Embedded Coder documentation
Control how Real-Time Workshop Embedded Coder generates function prototypes—arguments, argument order, and data types—for a model (for example, so the prototypes match the external code)	<i>EC only</i> — Configure function prototypes for the model by clicking the Configure Model Functions button on the →Real-Time Workshop > Interface pane of the Configuration Parameters dialog box and entering data in the Model Interface dialog box; alternatively, configure the function prototypes programmatically in the MATLAB command window or with a script	<ul style="list-style-type: none"> “Model Function Prototypes Example” “Controlling Generation of Function Prototypes” in the Real-Time Workshop Embedded Coder documentation

If You Need To...	Then...	For More Information, See...
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 Real-Time Workshop > Interface pane of the Configuration Parameters dialog box	<ul style="list-style-type: none"> Integrating the Generated Code into the External Environment Chapter 30, “Integrating External Code Using Model Configuration Parameters”
Generate code, which is to be integrated with an existing C code base, for a function-call or virtual subsystem	<i>EC only</i> — Review and adjust for exported subsystem requirements, configure the parent model to use an ERT target, and right-click build the subsystem block by using the Real-Time Workshop > Export Functions menu item	<ul style="list-style-type: none"> “Techniques for Exporting Function-Call Subsystems” <code>rtwdemo_export_functions</code> “Exporting Function-Call Subsystems” in the Real-Time Workshop Embedded Coder documentation

Exporting Algorithm Executables for System Simulation

If you have a Real-Time Workshop 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 “Creating and Using Host-Based Shared Libraries” in the Real-Time Workshop Embedded Coder documentation.

Making External Code Language Compatible With Generated Code

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();
#ifndef __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()
{
    ...
}
```


Integrating External Code Using Model Configuration Parameters

Configure a model such that the Real-Time Workshop code generator includes external code—headers, files and functions—in generated code by using the **Custom Code** pane.

Use the **Custom Code** pane to insert code into the generated files and to include additional files and paths in the build process.

To...	Select...
Insert custom code near the top of the generated <i>model.c</i> or <i>model.cpp</i> file, outside of any function	Source file and enter the custom code to insert.
Insert custom code near the top of the generated <i>model.h</i> file	Header file and enter the custom code to insert.

To...	Select...
Insert custom code inside the model's initialize function in the <i>model.c</i> or <i>model.cpp</i> file	Initialize function
Insert custom code inside the model's terminate function in the <i>model.c</i> or <i>model.cpp</i> file.	Terminate function and enter the custom code to insert. Also select the Terminate function required parameter on the Interface pane.
Add include directories, which contain header files, to the build process	Include directories and enter the absolute or relative paths to the directories. If you specify relative paths, the paths must be relative to the directory containing your model files, not relative to the build directory. The order in which you specify the directories 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. A file name is sufficient if the file is in the current MATLAB directory or in one of the include directories. For each additional source that you specify, the Real-Time Workshop build process expands a generic rule in the template makefile for the directory in which the source file is found. For example, if a source file is found in directory <i>inc</i> , the Real-Time Workshop build process adds a rule similar to the following:

```
% .obj: builddir\inc\%.c
        $(CC) -c -Fo$(@F) $(CFLAGS) $<
```

The Real-Time Workshop build process adds the rules in the order you list the source files.

To...	Select...
Add libraries to be linked	Libraries and enter the full paths or just the file names for the libraries. A file name is sufficient if the library is located in the current MATLAB directory or in one of the include directories.
Use the same custom code settings as those specified for simulation of Embedded MATLAB Function blocks, Stateflow charts, and Truth Table blocks	<p>Use the same custom code settings as Simulation Target</p> <p>Note This option refers to the Simulation Target pane in the Configuration Parameters dialog box.</p>
Enable a library model to use custom code settings unique from the parent model to which the library is linked	<p>Use local custom code settings (do not inherit from main model)</p> <p>Note This option is available only for library models that contain Embedded MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Select Tools > Open RTW Target in the Embedded MATLAB Editor or Stateflow Editor for your library model.</p>

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 descriptions of **Custom Code** pane parameters, see “Real-Time Workshop Pane: Custom Code” in the Real-Time Workshop reference documentation.

Integrating External Code Using Custom Code Blocks

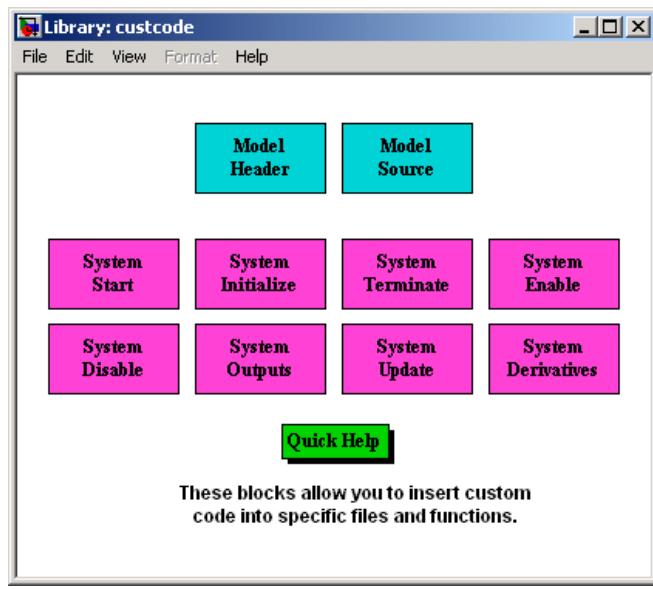
The following sections explain how to use blocks in the Custom Code block library to insert custom code into the code generated for a model. This chapter includes the following topics:

- “Custom Code Library” on page 31-2
- “Example: Using a Custom Code Block” on page 31-6
- “Custom Code in Subsystems” on page 31-9
- “Preventing User Source Code from Being Deleted from Build Directories” on page 31-10

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 Real-Time Workshop 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 Real-Time Workshop library. You can access the Real-Time Workshop library by using the Simulink Library Browser. You can access Custom Code blocks by using the Real-Time Workshop 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`.



This chapter discusses use of the Custom Code library only.

Note If you need to integrate custom C++ code with generated C code or vice versa, see Chapter 29, “About External Code Integration” for information on language compatibility requirements.

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 submodels (models referenced by Model blocks). However, when simulation targets for submodels are generated, all Custom Code blocks within them are ignored. On the other hand, when submodel code is generated to create Real-Time Workshop 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 Real-Time Workshop 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

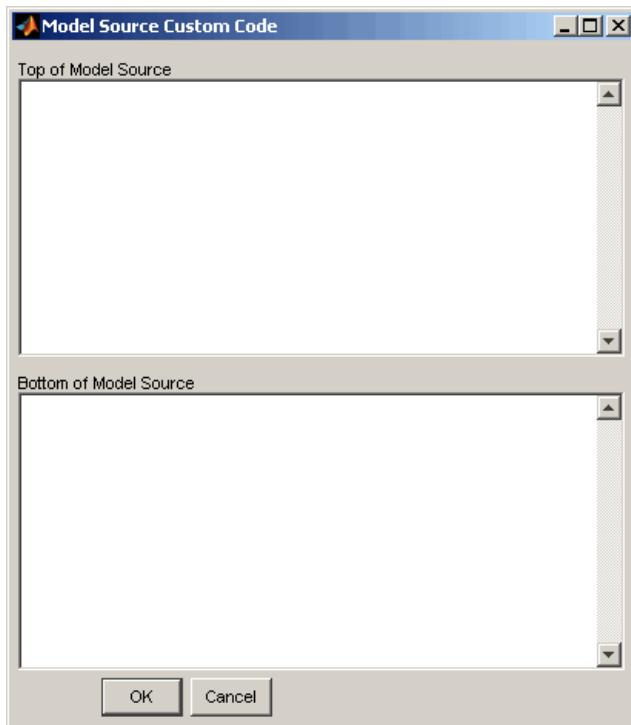
- *model.h* — Model Header File block
- *model.c* or *model.cpp* — Model Source File block

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.



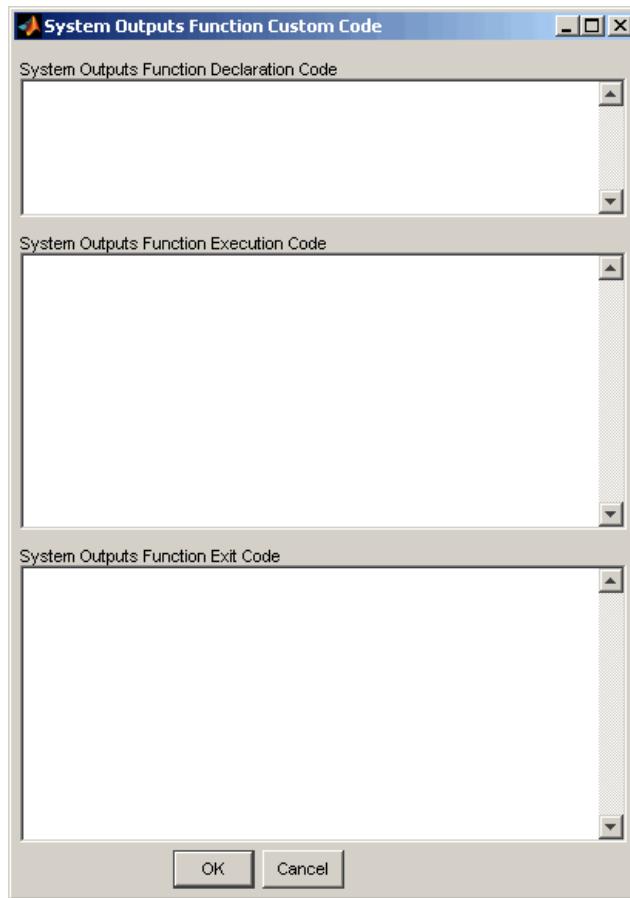
The eight function blocks in the second and third rows contain text fields to insert custom code sections at the top and bottom of these designated model functions:

- 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

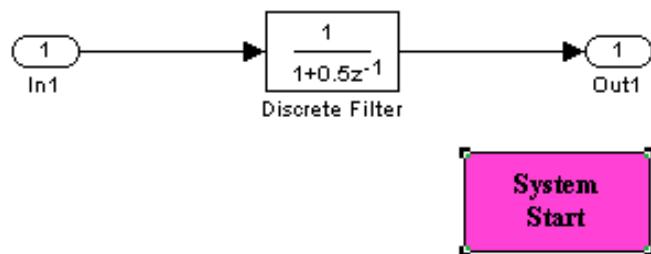
Each of these blocks provides a System Outputs Function Custom Code dialog box that contains three fields:

- Declaration code
- Execution code
- Exit code

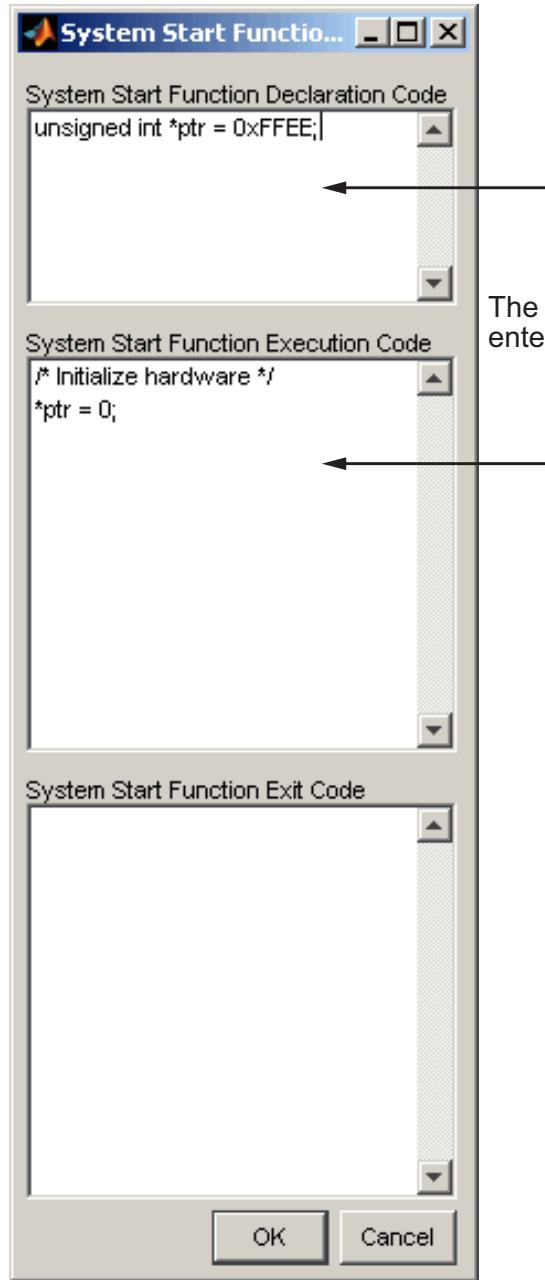


Example: Using a Custom Code Block

The following example uses a System Start Function block to introduce code into the Md1Start 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.



The code generator inserts the code entered here into the MdlStart function.

You can insert custom code into any or all of 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. There is no diagnostic setting to control this. To eliminate the error, remove the Custom Code block.

Preventing User Source Code from Being Deleted from Build Directories

Prior to Release 13 (Version 5.0), the Real-Time Workshop product did not delete any .c or .h files that the user had placed in the build directory when rebuilding targets. From Release 13 onward, all 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 directory, and you want to prevent the Real-Time Workshop 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 directory files with names having the pattern *model_**.c/.cpp (where * could be any string). The indenting is harmless, but can cause differences to be detected by source control software that might trigger unnecessary updates.

Integrating External Code Using S-Functions

- “Introduction to S-Functions for Code Generation” on page 32-2
- “Writing Noninlined S-Functions” on page 32-9
- “Writing Wrapper S-Functions” on page 32-12
- “Writing Fully Inlined S-Functions” on page 32-23
- “Automating Generation of Files for Fully Inlined S-Functions Using Legacy Code Tool” on page 32-25
- “Writing Fully Inlined S-Functions with the mdlRTW Routine” on page 32-30
- “Guidelines for Writing Inlined S-Functions” on page 32-56
- “Writing S-Functions That Support Expression Folding” on page 32-57
- “Writing S-Functions That Specify Port Scope and Reusability” on page 32-73
- “Writing S-Functions That Specify Sample Time Inheritance Rules” on page 32-79
- “Writing S-Functions That Support Code Reuse” on page 32-81
- “Writing S-Functions for Multirate Multitasking Environments” on page 32-82
- “Build Support for S-Functions” on page 32-90

Introduction to S-Functions for Code Generation

In this section...

- “About S-Functions” on page 32-2
- “Additional Information” on page 32-3
- “Classes of Problems Solved by S-Functions” on page 32-3
- “Types of S-Functions” on page 32-4
- “Basic Files Required for Implementation” on page 32-7
- “Guidelines for Writing S-Functions for Use with Real-Time Workshop Software” on page 32-8

About S-Functions

This chapter describes how to create S-functions that work seamlessly with Real-Time Workshop 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 chapter assumes that you understand the following concepts:

- Level 2 S-functions
- Target Language Compiler (TLC) scripting
- How the Real-Time Workshop software generates and builds C/C++ code

Note When this chapter 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 chapter uses the abbreviation TLC.

Note The guidelines presented in this chapter are for Real-Time Workshop users. Even if you do not currently use the Real-Time Workshop code generator, you should follow the practices presented in this chapter when writing S-functions, especially if you are creating general-purpose S-functions.

Additional Information

See the Target Language Compiler documentation and other Real-Time Workshop documentation for more information on the code generation process.

See “Inlining S-Functions” in the Target Language Compiler documentation for additional information on inlining 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 Real-Time Workshop products. These problems include

- Extending the set of algorithms (blocks) provided by the Simulink and Real-Time Workshop products
- Interfacing legacy (hand-written) code with the Simulink and Real-Time Workshop 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 Real-Time Workshop 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 chapter, you can create S-functions for most applications that work with the Real-Time Workshop 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 Real-Time Workshop 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 chapter discusses the typical problems that you may face and how to create S-functions for applications that need to work with the Simulink and Real-Time Workshop 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 Real-Time Workshop 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 Real-Time Workshop 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 Real-Time Workshop products.”

- 3** “I want to implement a highly optimized algorithm in the Simulink and Real-Time Workshop products that looks like a built-in block and generates very efficient code.”

The 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 Real-Time Workshop generated code. In general, you implement your algorithm once according to the S-function API. The Simulink engine and Real-Time Workshop generated code call the S-function routines (for example, `mdlOutputs`) at the appropriate points 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 Simulink models and Real-Time Workshop 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 any TLC coding. Noninlined S-functions are the default case for the Real-Time Workshop 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 **Real-Time Workshop** 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 “Writing Noninlined S-Functions” on page 32-9.

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 correctly in the Simulink environment, a certain amount of overhead code is necessary. When the Real-Time Workshop 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 all 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 Real-Time Workshop 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 appropriate 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 Real-Time Workshop 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 Real-Time Workshop 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 “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink Writing S-Functions documentation and “Automating

Generation of Files for Fully Inlined S-Functions Using Legacy Code Tool” on page 32-25.

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 chapter 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 Real-Time Workshop 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 for Use with Real-Time Workshop Software

- Use only C MEX S-functions with the Real-Time Workshop code generator. You cannot use Level-1 MATLAB language S-functions with Real-Time Workshop 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 needed to compile and build the S-function for simulation and generate a masked S-function block configured to call existing external code. For more information, see “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation and “Automating Generation of Files for Fully Inlined S-Functions Using Legacy Code Tool” on page 32-25.
- If you are rapid prototyping, inlining an S-function might not be necessary. If you choose not to inline the C MEX S-function, write the S-function, include it directly in the model, and let the Real-Time Workshop software generate the code. For more information, see “Writing Noninlined S-Functions” on page 32-9.

Writing Noninlined S-Functions

In this section...

[“About Noninlined S-Functions” on page 32-9](#)

[“Guidelines for Writing Noninlined S-Functions” on page 32-9](#)

[“Noninlined S-Function Parameter Type Limitations” on page 32-10](#)

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 correctly. 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 Real-Time Workshop build process with the names of all modules used to build the S-function. You can do this by using the Real-Time Workshop template make file or the `set_param` function. 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 all the required modules:

```
set_param(sfun_block, "SFunctionModules","sfun_module1 sfun_module2")
```

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_matrix.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`.

Writing Wrapper S-Functions

In this section...

- “About Wrapper S-Functions” on page 32-12
- “MEX S-Function Wrapper” on page 32-12
- “TLC S-Function Wrapper” on page 32-17
- “The Inlined Code” on page 32-22

About Wrapper S-Functions

This section describes how to create S-functions that work seamlessly with the Simulink and Real-Time Workshop 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 Real-Time Workshop generated code with little or no change to your original C/C++ function. A *MEX S-function wrapper* is an S-function that calls code that resides in another module. A *TLC S-function wrapper* is a TLC file that specifies how the code generator should call your code (the same code that was called from the C MEX S-function wrapper).

Note A MEX S-function wrapper must only be used in the MATLAB version in which the wrapper is created.

Suppose you have an algorithm (that is, a C function) called `my_alg` that resides in the file `my_alg.c`. You can integrate `my_alg` into a Simulink model by creating a MEX S-function wrapper (for example, `wrapsfcn.c`). Once this is done, a Simulink model can call `my_alg` from an S-Function block.

However, the Simulink S-function contains a set of empty functions that the Simulink engine requires for various API-related purposes. For example, although only `mdlOutputs` calls `my_alg`, the engine calls `mdlTerminate` as well, even though this S-function routine performs no action.

You can integrate `my_alg` into generated code (that is, embed the call to `my_alg` in the generated code) by creating a TLC S-function wrapper (for example, `wrapsfcn.tlc`). The advantage of creating a TLC S-function wrapper is that the empty function calls can be eliminated and the overhead of executing the `mdlOutputs` function and then the `my_alg` function can be eliminated.

Wrapper S-functions are useful when you are creating new algorithms that are procedural in nature or when you are integrating legacy code into a Simulink model. However, if you want to create code that is

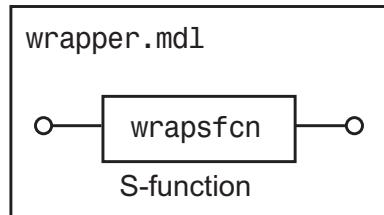
- Interpretive in nature (that is, highly parameterized by operating modes)
- Heavily optimized (that is, no extra tests to decide what mode the code is operating in)

then you must create a *fully inlined TLC file* for your S-function.

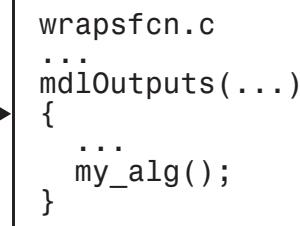
The next figure shows the wrapper S-function concept.

Simulink

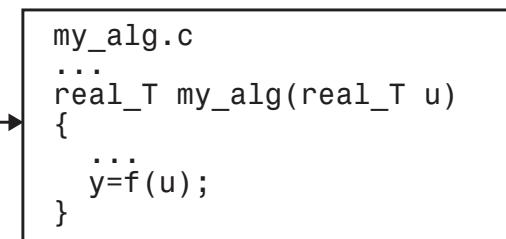
Place the name of your s-function in the S-Function block's dialog box.



In Simulink, the S-function calls `mdlOutputs`, which in turn calls `my_alg`.



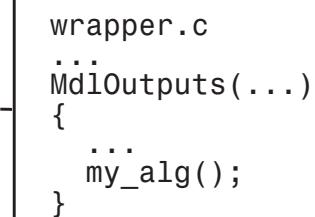
`mdlOutputs` in `wrapsfcn.mex` calls external function `my_alg`.



*See note below

Real-Time Workshop

`wrapper.c`, the generated code, calls `mdlOutputs`, which then calls `my_alg`.

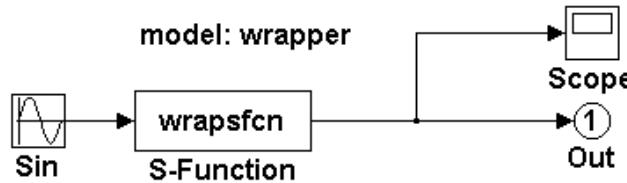


In the TLC wrapper version of the S-function, `mdlOutputs` in `wrapper.exe` calls `my_alg`.

*The dotted line is the path taken if the S-function does not have a TLC wrapper file. If there is no TLC wrapper file, the generated code calls `mdlOutputs`.

Using an S-function wrapper to import algorithms in your Simulink model means that the S-function serves as an interface that calls your C/C++ algorithms from `mdlOutputs`. S-function wrappers have the advantage that you can quickly integrate large standalone C/C++ programs into your model without having to make changes to the code.

The following sample model includes an S-function wrapper.



There are two files associated with the `wrapsfcn` block, the S-function wrapper and the C/C++ code that contains the algorithm. The S-function wrapper code for `wrapsfcn.c` appears below. The first three statements do the following:

- 1** Defines the name of the S-function (what you enter in the Simulink S-Function block dialog).
- 2** Specifies that the S-function is using the level 2 format.
- 3** Provides access to the `SimStruct` data structure, which contains pointers to data used during simulation and code generation and defines macros that store data in and retrieve data from the `SimStruct`.

For more information, see “Templates for C S-Functions” in the Simulink documentation.

```

#define S_FUNCTION_NAME wrapsfcn
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"

extern real_T my_alg(real_T u); /* Declare my_alg as extern */

/*
 * mdlInitializeSizes - initialize the sizes array
 */
static void mdlInitializeSizes(SimStruct *S)
{

    ssSetNumSFcnParams( S, 0); /*number of input arguments*/

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 1);
  
```

```
ssSetInputPortDirectFeedThrough(S, 0, 1);

if (!ssSetNumOutputPorts(S,1)) return;
ssSetOutputPortWidth(S, 0, 1);

ssSetNumSampleTimes( S, 1);
}

/*
 * mdlInitializeSampleTimes - indicate that this S-function runs
 * at the rate of the source (driving block)
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

/*
 * mdlOutputs - compute the outputs by calling my_alg, which
 * resides in another module, my_alg.c
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    real_T          *y     = ssGetOutputPortRealSignal(S,0);
    *y = my_alg(*uPtrs[0]); /* Call my_alg in mdlOutputs */
}

/*
 * mdlTerminate - called when the simulation is terminated.
 */
static void mdlTerminate(SimStruct *S)
{

#endif 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 22-4 in the Real-Time Workshop 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 Real-Time Workshop 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)
{
    /* (no start code 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)
{
    /* (no update code 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

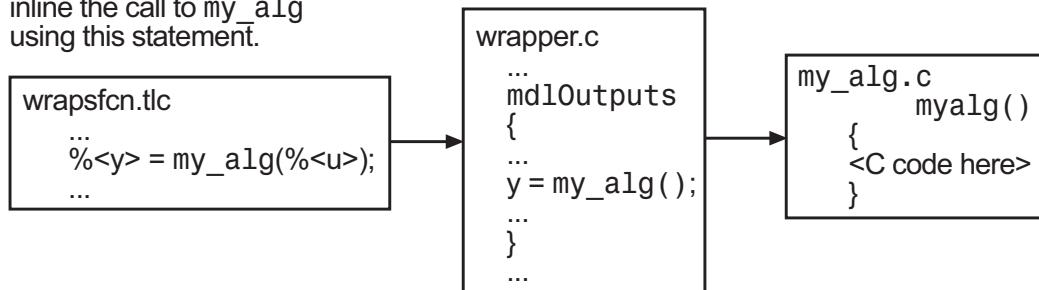
The generated code makes the call to your S-function, `wrapsfcn.c`, in `mdlOutputs` by using this code:

```
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 documentation for details). The TLC file must contain the function call to `my_alg`. The following figure shows the relationships between the algorithm, the wrapper S-function, and the `sfunction.tlc` file.

The `wrapsfcn.tlc` file tells Real-Time Workshop how to inline the call to `my_alg` using this statement.



To inline this call, you have to place your function call in an `sfunction.tlc` file with the same name as the S-function (in this example, `wrapsfcn.tlc`). This causes the Target Language Compiler to override the default method of placing calls to your S-function in the generated code.

This is the `wrapsfcn.tlc` file that inlines `wrapsfcn.c`.

```
%% File      : wrapsfcn.tlc
%% Abstract:
%%           Example inlined tlc file for S-function wrapsfcn.c
%%
```

```
%implements "wrapsfcn" "C"

%% Function: BlockTypeSetup =====
%% Abstract:
%%     Create function prototype in model.h as:
%%     "extern real_T my_alg(real_T u);"
%%
%function BlockTypeSetup(block, system) void
    %openfile buffer
        extern real_T my_alg(real_T u); /* This line is placed in wrapper.h */
    %closefile buffer
    %<LibCacheFunctionPrototype(buffer)>
%endfunction %% BlockTypeSetup

%% Function: Outputs =====
%% Abstract:
%%     y = my_alg( u );
%%
%function Outputs(block, system) Output
    /* %<Type> Block: %<Name> */
    %assign u = LibBlockInputSignal(0, "", "", 0)
    %assign y = LibBlockOutputSignal(0, "", "", 0)
    %% PROVIDE THE CALLING STATEMENT FOR "algorithm"
    %% The following line is expanded and placed in mdlOutputs within wrapper.c
    %<y> = my_alg(%<u>);

%endfunction %% Outputs
```

The first section of this code inlines the `wrapsfcn` S-Function block and generates the code in C:

```
%implements "wrapsfcn" "C"
```

The next task is to tell the code generator that the routine `my_alg` needs to be declared external in the generated `wrapper.h` file for any `wrapsfcn` S-Function blocks in the model. You only need to do this once for all `wrapsfcn` S-Function blocks, so use the `BlockTypeSetup` function. In this function, you tell the Target Language Compiler to create a buffer and cache the `my_alg` as `extern` in the `wrapper.h` generated header file.

The final step is the inlining of the call to the function `my_alg`. This is done by the `Outputs` function. In this function, you access the block's input and output and place a direct call to `my_alg`. The call is embedded in `wrapper.c`.

The Inlined Code

The code generated when you inline your wrapper S-function is similar to the default generated code. The `mdlTerminate` function no longer contains 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` no longer creates a child `SimStruct` for the S-function because the generated code is calling `my_alg` directly. This eliminates over 1 KB of memory usage.

Writing 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 Example

A more advanced multiport inlined S-function example is `sfun_multiport.c` and `matlabrootsfun_multiport.tlc`. This S-function demonstrates 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.

Automating Generation of Files for Fully Inlined S-Functions Using Legacy Code Tool

In this section...

“Legacy Code Tool and Code Generation” on page 32-25

“Generating Inlined S-Function Files for Code Generation Support” on page 32-26

“Applying Model Code Style Settings to Legacy Functions” on page 32-27

“Addressing Dependencies on Files in Different Locations” on page 32-28

“Deploying Generated S-Functions for Simulation and Code Generation” on page 32-29

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 directories other than the directory 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 Real-Time Workshop build process looks for the generated `rtwmakecfg.m` file in the same directory as the S-function's dynamically loadable executable and calls the `rtwmakecfg` function if the software finds the file.

For more information, see “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink Writing S-Functions documentation.

Generating Inlined S-Function Files for Code Generation Support

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 (1) before generating the S-function source file from your existing C function. For example:

```
def.Options.singleCPPMexFile = 1;
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

ans =

0
legacy_code('sfcn_cmex_generate', def);
legacy_code('sfcn_tlc_generate', def);
```

singleCPPMexFile Limitations

You cannot set the `singleCPPMexFile` field to true (1) if:

- You set `Options.language='C++'`.
- You use one of the following Simulink objects with the `IsAlias` property set to true (1):
 - `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
- The `HeaderFiles` field of the Legacy Code Tool structure specifies multiple header files

Applying Model Code Style Settings to Legacy Functions

To apply the code style specified by a model's configuration parameters 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 (1). For example:

```
def.Options.singleCPPMexFile = 1;
```

To verify the setting, enter:

```
def.Options.singleCPPMexFile
```

singleCPPMexFile Limitations

You cannot set the `singleCPPMexFile` field to true (1) if:

- You set `Options.language='C++'`.
- You use one of the following Simulink objects with the `IsAlias` property set to true (1):
 - `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
- The `HeaderFiles` field of the Legacy Code Tool structure specifies multiple header files

Addressing Dependencies on Files in Different Locations

By default, the Legacy Code Tool assumes that all files on which an S-function depends reside in the same directory 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 Real-Time Workshop 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 directory 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 Writing S-Functions documentation

For example, if you define `defs` as an array of Legacy Code Tool structures, you call `legacy_code` with '`rtwmakecfg_generate`' once.

```
defs = [defs1();defs2();defs3()];
```

```
legacy_code('rtwmakecfg_generate', defS);
```

For more information, see “Build Support for S-Functions” on page 32-90.

Deploying Generated 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
- All 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 directory that contains the S-function files to the MATLAB path.
- If the Legacy Code Tool data structure registers any required files as absolute paths and the location of the files changes, they must regenerate the `rtwmakecfg.m` file.

Writing Fully Inlined S-Functions with the `mdlRTW` Routine

In this section...

“About S-Functions and `mdlRTW`” on page 32-30

“S-Function RTWdata” on page 32-31

“The Direct-Index Lookup Table Algorithm” on page 32-32

“The Direct-Index Lookup Table Example” on page 32-33

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 performance 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 demonstrates how to create a 1-D lookup S-function, `sfun_directlook.c`, and its corresponding inlined `sfun_directlook.tlc` file. (See the Target Language Compiler documentation for more details on the Target Language Compiler.) This 1-D direct-index lookup table example demonstrates 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 Real-Time Workshop 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 .mdl 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.

The 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 is not needed in all situations. In this example, you create a lookup table that directly indexes the output vector (*y*-data vector) based on the current input (*x*-data) point.

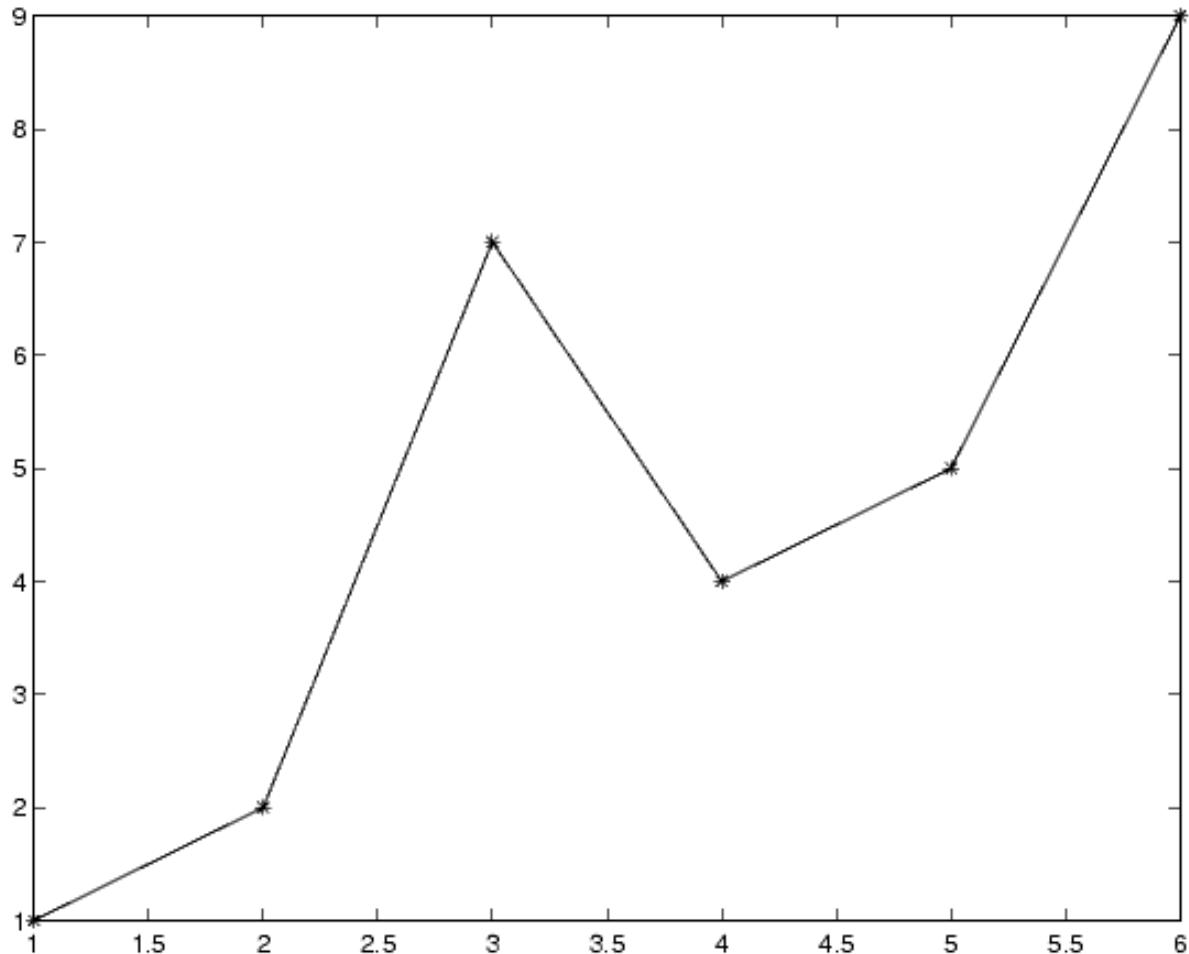
This direct 1-D lookup example computes an approximate solution $p(x)$ to a partially known function $f(x)$ at $x=x_0$, given data point pairs (x,y) in the form of an *x*-data vector and a *y*-data vector. For a given data pair (for example, the *i*'th pair), $y_i = f(x_i)$. It is assumed that the *x*-data values are monotonically increasing. If x_0 is outside the range of the *x*-data vector, the first or last point is returned.

The parameters to the S-function are

XData, YData, XEvenlySpaced

XData and YData are double vectors of equal length representing the values of the unknown function. XDataEvenlySpaced is a scalar, 0.0 for false and 1.0 for true. If the XData vector is evenly spaced, XDataEvenlySpaced is 1.0 and more efficient code is generated.

The following graph shows how the parameters XData=[1:6] and YData=[1,2,7,4,5,9] are handled. For example, if the input (*x*-value) to the S-Function block is 3, the output (*y*-value) is 7.



The 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, `matlabroot/simulink/src/sfun_directlook.c`,

and a corresponding `matlabroot/simulink/src/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 demonstrates 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 `matlabroot/toolbox/simulink/blocks/tlc_c/sfun_directlook.tlc` (see the example in “mdlRTW Usage” on page 32-35).

Error Handling

In this example, the `mdlCheckParameters` routine verifies that

- The new parameter settings are correct.
- `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 if needed.

The `mdlInitializeSizes` function explicitly calls `mdlCheckParameters` after it verifies that the number of parameters passed to the S-function is correct. 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. This increases performance.

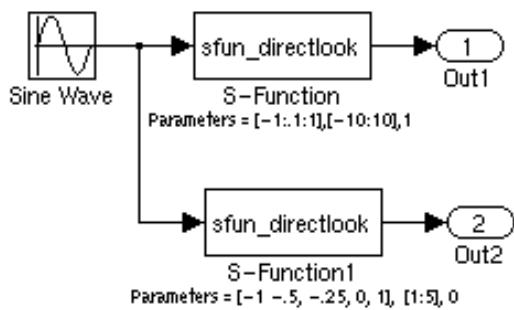
mdlRTW Usage

The Real-Time Workshop 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 Real-Time Workshop build process that the module `lookup_index.c` is needed when creating the executable.

When generating code for this model, the Real-Time Workshop 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

```

/*
 * sfun_directlook_ex.c
 *
 * Real-Time Workshop code generation for Simulink model
 * "sfun_directlook_ex.mdl".
 *
 * Model Version : 1.2

```

```

* Real-Time Workshop version : 6.0 (R14) 06-Apr-2004
* C source code generated on : Fri Apr 09 09:15:12 2004
*/

#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.clockTickHO;
    sfun_directlook_ex_M->Timing.t[0] = sfun_directlook_ex_M->Timing.clockTick0 *
        sfun_directlook_ex_M->Timing.stepSize0 +
        sfun_directlook_ex_M->Timing.clockTickHO *
        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/simulink/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. No
*      interpolation is 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.  
* $Revision: 1.1.4.10 $  
*/  
  
#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 Real-Time Workshop. However,  
 * this S-function does work with Real-Time Workshop via  
 * the Target Language Compiler technology. See  
 * matlabroot/toolbox/simulink/blocks/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 *
 *=====
#endif
#ifndef TRUE
#define TRUE 1
#endif
#ifndef 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 xlabel, 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 that the new parameter setting are correct.
*
*      You should add a call to this routine from mdlInitializeSizes
*      to check the parameters. After setting your sizes elements, you should:
*          if (ssGetSFnParamsCount(S) == ssGetNumSFnParams(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.
     */
#ifndef 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_REUSEABLE_AND_LOCAL);
ssSetInputPortOverWritable(S, 0, TRUE);

```

```
    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);

    ssSetOutputPortOptimOpts(S, 0, SS_REUSEABLE_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 */
```

```

#ifndef 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 Real-Time Workshop 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 correponding .tlc file so as to work with RTW. You will find
 *   the sfun_directlook.tlc in <matlaroot>/toolbox/simulink/blocks/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,
                                SSSWRITE_VALUE_QSTR,
                                "XSpacing",
                                even ? "EvenlySpaced" : "UnEvenlySpaced")){


```

```
        return; /* An error occurred which will be reported by Simulink */
    }
}
#endif /* MDL_RTW */

/*=====
 * Required S-function trailer
 *=====
#endif /* 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_directlookup.c */
```

matlabroot/simulink/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.
 *      $Revision: 1.1.4.10 $
 */
#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 appropriate 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/blocks/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.
%% $Revision: 1.1.4.10 $
```

```
%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 maxlen, 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.
%%     No interpolation is performed.
%%
%%     The S-function parameters are:
%%         XData
%%         YData
%%         XEvenlySpaced: 0 or 1
%%     The third parameter cannot be changed during execution and is
%%     written to the model.rtw file in XSpacing filed of the SFcnParamSettings
```

```
%%      record as "EvenlySpaced" or "UnEvenlySpaced". The first two parameters
%%      can change during execution and show up in the parameter vector.
%%
%function Outputs(block, system) Output
/* %<Type> Block: %<Name> */
{
    %assign rollVars = [ "U", "Y" ]
    %%
    %% Load XData and YData as local variables
    %%
    const real_T *xData  = %<LibBlockParameterAddr(XData, "", "", 0)>;
    const real_T *yData  = %<LibBlockParameterAddr(YData, "", "", 0)>;
    %assign xDataLen = SIZE(XData.Value, 1)
    %%
    %% When the XData is evenly spaced, we use the direct lookup algorithm
    %% to locate the lookup index.
    %%
    %if SFcnParamSettings.XSpacing == "EvenlySpaced"
        real_T spacing = xData[1] - xData[0];

        %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
        %assign u = LibBlockInputSignal(0, "", lcv, idx)
        %assign y = LibBlockOutputSignal(0, "", lcv, idx)
        if ( %<u> <= xData[0] ) {
            %<y> = yData[0];
        } else if ( %<u> >= yData[%<xDataLen-1>] ) {
            %<y> = yData[%<xDataLen-1>];
        } else {
            int_T idx = (int_T)( ( %<u> - xData[0] ) / spacing );
            %<y> = yData[idx];
        }
        %%
        %% Generate an empty line if we are not rolling,
        %% so that it looks nice in the generated code.
        %%
        %if lcv == ""
            %endif
            %endroll
        %else
```

```
%% When the XData is unevenly spaced, we use a bisection search to
%% locate the lookup index.
int_T idx;

%assign xDataAddr = LibBlockParameterAddr(XData, "", "", 0)
%roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %assign u = LibBlockInputSignal(0, "", lcv, idx)
    idx = GetDirectLookupIndex(xData, %<xDataLen>, %<u>);
    %assign y = LibBlockOutputSignal(0, "", lcv, idx)
    %<y> = yData[idx];
    %%
    %% Generate an empty line if we are not rolling,
    %% so that it looks nice in the generated code.
    %%
%if lcv == ""

    %endif
%endroll
%endif
}
%endfunction
%% EOF: sfun_directlook.tlc
```

Guidelines for Writing Inlined S-Functions

- Consider using the block property `RTWdata` (see “S-Function RTWdata” on page 32-31). 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 Real-Time Workshop 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

Writing S-Functions That Support Expression Folding

In this section...

“Introduction” on page 32-57

“Categories of Output Expressions” on page 32-58

“Acceptance or Denial of Requests for Input Expressions” on page 32-63

“Utilizing Expression Folding in Your TLC Block Implementation” on page 32-67

Introduction

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 “Overview of S-Functions” in the Simulink Writing S-Functions documentation).
- The Target Language Compiler (see the Target Language Compiler documentation)

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 there are multiple output destinations.

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 32-58).

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 need to understand when it is appropriate to request acceptance and generation of expressions for specific blocks. It is not necessary for you 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*.

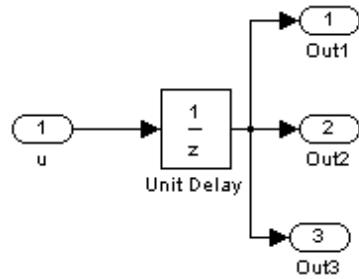
A *constant* output expression is a direct access to one of the block’s parameters. For example, the output of a Constant block is defined as a constant expression because the output expression is simply a direct access to the block’s Value parameter.

A *trivial* output expression is an expression that can be repeated, without any 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 involves no 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, involving no 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;
}
  
```

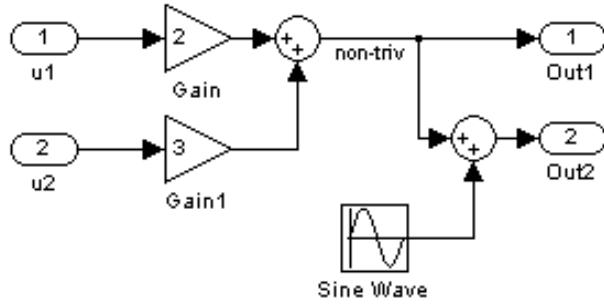
```

/* 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 Md1Outputs(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
     *   Import: <Root>/u1
     *   Gain: <Root>/Gain1
     *   Import: <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);
}

```

Specifying the Category of an Output Expression

The S-Function API provides macros that let you declare whether an output of a block should be an expression, and if so, to specify the category of the expression. The following table specifies when to declare a block output to be a constant, trivial, or generic output expression.

Types of Output Expressions

Category of Expression	When to Use
Constant	Use only if block output is a direct memory access to a block parameter.
Trivial	Use only if block output is an expression that can appear multiple times in the code without reducing efficiency (for example, a direct memory access to a field of the DWork vector, or a literal).
Generic	Use if output is an expression, but not constant or trivial.

You must declare outputs as expressions in the `mdlSetWorkWidths` function using macros defined in the S-Function API. The macros have the following arguments:

- `SimStruct *S`: pointer to the block's `SimStruct`.
- `int idx`: zero-based index of the output port.
- `bool value`: pass in `TRUE` if the port generates output expressions.

The following macros are available for setting an output to be a constant, trivial, or generic expression:

- `void ssSetOutputPortConstOutputExprInRTW(SimStruct *S, int idx, bool value)`
- `void ssSetOutputPortTrivialOutputExprInRTW(SimStruct *S, int idx, bool value)`

- `void ssSetOutputPortOutputExprInRTW(SimStruct *S, int idx, bool value)`

The following macros are available for querying the status set by any 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 “Generic Conditions for Denial of Requests to Output Expressions” on page 32-67.

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 any block that is connected to that input port is not permitted to output a generic or trivial expression.

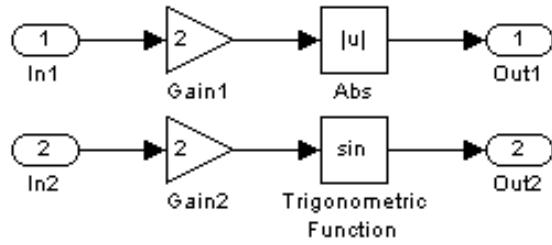
A request to output a constant expression is never denied, because there is no performance penalty for a constant expression, and it is always possible to take the parameter's address.

Example: Acceptance and Denial of Expressions at Block Inputs

This example shows how various built-in blocks handle requests to accept different categories of expressions at their inputs.

The following sample model contains

- Two Gain blocks. Gain blocks request their destination blocks to accept generic expressions.
- An Abs block. This block always denies expressions at its input port. The Abs block code uses the macro `rt_ABS(u)`, which evaluates the input `u` twice. (See the TLC implementation of the Abs block in `matlabroot/rtw/c/tlc/blocks/absval.tlc`.)
- A Trigonometric Function block. This block accepts expressions at its input port.



The Gain1 block request to output an expression is denied by the Abs block. The Gain2 block request to output an expression is accepted by the Trigonometric Function block.

The generated code is shown in the code excerpt below. The output of the Gain1 block is stored in the temporary variable `rtb_Gain1`, rather than generating an input expression to the Abs block.

```

void MdlOutputs(int_T tid)
{
    /* local block i/o variables */
    real_T rtb_Gain1;

    /* Gain: '<Root>/Gain1' incorporates:
     *   Import: '<Root>/In1'
     *
     * Regarding '<Root>/Gain':
     *   Gain value: 2.0
     */
    rtb_Gain1 = rtU.In1 * 2.0;

    /* Outport: '<Root>/Out1' incorporates:
     *   Abs: '<Root>/Abs'
     */
    rtY.Out1 = rt_ABS(rtb_Gain1);

    /* Outport: '<Root>/Out2' incorporates:
     *   Trigonometry: '<Root>/Trigonometric Function'
     *   Gain: '<Root>/Gain2'
     */
}

```

```
*      Import: '<Root>/In2'  
*  
* Regarding '<Root>/Gain2':  
*   Gain value: 2.0  
*/  
rtY.Out2 = sin((2.0 * rtU.In2));  
}
```

Using the 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)
```

Generic Conditions for Denial of 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.

Utilizing Expression Folding in Your 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 “Using the S-Function API to Specify Input Expression Acceptance” on page 32-66.
- Output ports, as explained in “Categories of Output Expressions” on page 32-58.

This section discusses required modifications to the TLC implementation.

Expression Folding Compliance

In the `BlockInstanceSetup` function of your S-function, you must ensure that your block registers that it is compliant with expression folding. If you fail to do this, any 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)
```

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 Real-Time Workshop 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.

Outputting 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 an appropriate 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 appropriate 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, y_0 , is equal to two times the first input.
- The second output, y_1 , 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
%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, all 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)>
```

Writing 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:

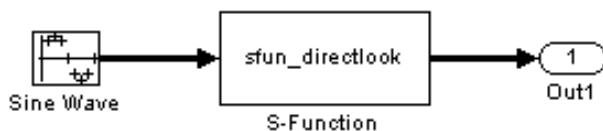
- `SS_NOT_REUSEABLE_AND_GLOBAL`: Indicates that the input and output ports are stored in separate memory locations in the global block input and output structure
- `SS_NOT_REUSEABLE_AND_LOCAL`: Indicates that the Real-Time Workshop software can declare individual local variables for the input and output ports
- `SS_REUSEABLE_AND_LOCAL`: Indicates that the Real-Time Workshop software can reuse a single local variable for these input and output ports
- `SS_REUSEABLE_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 guarantee the Real-Time Workshop 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 effect the generated code. The following model contains an S-function block pointing to the C MEX S-function `matlabroot/simulink/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_REUSEABLE_AND_LOCAL);
ssSetInputPortOverWritable(S, 0, TRUE);

/* snip */
ssSetOutputPortOptimOpts(S, 0, SS_REUSEABLE_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 <code>mdlInitializeSizes</code> code	Generated code
Inputs: Local, reusable, overwritable Outputs: Local, reusable	<pre> ssSetInputPortOptimOpts(S, 0, SS_REUSEABLE_AND_LOCAL); ssSetInputPortOverWritable(S, 0, TRUE); ssSetOutputPortOptimOpts(S, 0, SS_REUSEABLE_AND_LOCAL); </pre>	<p>The <i>model.c</i> file declares a local variable in the output function.</p> <pre> /* local block i/o variables */ real_T rtb_SFunction[2]; </pre>
Inputs: Global, reusable, overwritable Outputs: Global, reusable	<pre> ssSetInputPortOptimOpts(S, 0, SS_REUSEABLE_AND_GLOBAL); ssSetInputPortOverWritable(S, 0, TRUE); </pre>	<p>The <i>model.h</i> file defines a block signals structure with a single element to store the S-function's input and output.</p> <pre> /* Block signals (auto storage) */ </pre>

Scope and reusability	S-function <code>mdlInitializeSizes</code>	Generated code
	<pre>ssSetOutputPortOptimOpts(S, 0, SS_REUSEABLE_AND_GLOBAL);</pre>	<pre>typedef struct { real_T SFunction[2]; } BlockIO_sl_directlook;</pre> <p>The <code>model.c</code> file uses this element of the structure in calculations of the S-function's input and output signals.</p> <pre>/* 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]</pre>
Inputs: Local, not reusable Outputs: Local, not reusable	<pre>ssSetInputPortOptimOpts(S, 0, SS_NOT_REUSEABLE_AND_LOCAL); ssSetInputPortOverWritable(S, 0, FALSE); ssSetOutputPortOptimOpts(S, 0, SS_NOT_REUSEABLE_AND_LOCAL);</pre>	<p>The <code>model.c</code> file declares local variables for the S-function's input and output in the output function</p> <pre>/* local block i/o variables */ real_T rtb_SineWave[2]; real_T rtb_SFunction[2];</pre>
Inputs: Global, not reusable Outputs: Global, not reusable	<pre>ssSetInputPortOptimOpts(S, 0, SS_NOT_REUSEABLE_AND_GLOBAL); ssSetInputPortOverWritable(S, 0, FALSE); ssSetOutputPortOptimOpts(S, 0, SS_NOT_REUSEABLE_AND_GLOBAL);</pre>	<p>The <code>model.h</code> file defines a block signal structure with individual elements to store the S-function's input and output.</p> <pre>/* Block signals (auto storage) */ typedef struct { real_T SineWave[2]; real_T SFunction[2]; } BlockIO_sl_directlook;</pre> <p>The <code>model.c</code> file uses the different elements in this structure when</p>

Scope and reusability	S-function <code>mdlInitializeSizes</code>	Generated code
		<p>calculating the S-function's input and output.</p> <pre>/* Sin: '<Root>/Sine Wave' */ sl_directlook_B.SineWave[0] = sin ... /* snip */ /*S-Function Block:<Root>/S-Function*/ { const real_T *xData = &sl_directlook_P.SFunction_XData[0]</pre>

Writing S-Functions That Specify Sample Time Inheritance Rules

For the Simulink engine to accurately determine whether a model can inherit a sample time, the S-functions in the model need to specify how they use sample times. You can specify this information by calling the macro `ssSetModelReferenceSampleTimeInheritanceRule` from `mdlInitializeSizes` or `mdlSetWorkWidths`. To use this macro:

- 1** Check whether the S-function calls any of the following macros:
 - `ssGetSampleTime`
 - `ssGetInputPortSampleTime`
 - `ssGetOutputPortSampleTime`
 - `ssGetInputPortOffsetTime`
 - `ssGetOutputPortOffsetTime`
 - `ssGetSampleTimePtr`
 - `ssGetInputPortSampleTimeIndex`
 - `ssGetOutputPortSampleTimeIndex`
 - `ssGetSampleTimeTaskID`
 - `ssGetSampleTimeTaskIDPtr`
- 2** Check for the following in your S-function TLC code:
 - `LibBlockSampleTime`
 - `CompiledModel.SampleTime`
 - `LibBlockInputSignalSampleTime`
 - `LibBlockInputSignalOffsetTime`
 - `LibBlockOutputSignalSampleTime`
 - `LibBlockOutputSignalOffsetTime`
- 3** Depending on your search results, use `ssSetModelReferenceSampleTimeInheritanceRule` as indicated in the following table.

If...	Use...
None of the macros or functions are present, the S-function does not preclude the model from inheriting a sample time.	<code>ssSetModelReferenceSampleTimeInheritanceRule (S, USE_DEFAULT_FOR_DISCRETE_INHERITANCE)</code>
Any of the macros or functions are used for <ul style="list-style-type: none"> • Throwing errors if sample time is inherited, continuous, or constant • Checking <code>ssIsSampleHit</code> • Checking whether sample time is inherited in either <code>mdlSetInputPortSampleTime</code> or <code>mdlSetOutputPortSampleTime</code> before setting 	<code>ssSetModelReferenceSampleTimeInheritanceRule... (S, USE_DEFAULT_FOR_DISCRETE_INHERITANCE)</code>
The S-function uses its sample time for computing parameters, outputs, and so on	<code>ssSetModelReferenceSampleTimeInheritanceRule (S, DISALLOW_SAMPLE_TIME_INHERITANCE)</code>

Note If an S-function does not set the `ssSetModelReferenceSampleTimeInheritanceRule` macro, by default the Simulink engine assumes that the S-function does not preclude the model containing that S-function from inheriting a sample time. However, the engine issues a warning indicating that the model includes S-functions for which this macro is not set.

You can use settings on the **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**.

Writing S-Functions That Support Code Reuse

The Real-Time Workshop *code reuse* feature generates code for a subsystem in the form of a single function that is invoked wherever the subsystem occurs in the model (see “Creating Subsystems” on page 3-37). If a subsystem contains S-functions, the S-functions must be compatible with the code reuse feature. Otherwise, the code generator might not generate reusable code from the subsystem or might generate incorrect code.

If you want your S-function to support the subsystem code reuse feature, you must ensure that the S-function meets 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 assures the code generator that your S-function meets the requirements for subsystem code reuse.

Writing S-Functions for Multirate Multitasking Environments

In this section...

“Introduction” on page 32-82

“Rate Grouping Support in S-Functions” on page 32-82

“Creating Multitasking-Safe, Multirate, Port-Based Sample Time S-Functions” on page 32-83

Introduction

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 Real-Time Workshop 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 substrate 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 Real-Time Workshop Embedded Coder documentation.

For each multirate S-function that is not rate grouping-compliant, the Real-Time Workshop software issues the following warning when you build:

```
Warning: Real-Time Workshop: 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, as appropriate.

Creating Multitasking-Safe, 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. Any 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 any 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 always safely 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 always the value when both the faster and slower rate had their hits (and possible earlier input data as well, depending on the algorithm). Any 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: $y = u(t_{slow})$
- 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: $y = \text{average}(u)$

- Input: local and reusable
- Output: local and reusable
- DirectFeedthrough: yes

(Assume DWork[0:10] and DWork[mycounter] are initialized to zero)

```
OutputFcn
if (ssIsSampleHit(".1")) {
    /* In general, processing on 'u' could be done here,
       it runs on every hit of the fast rate. */
    DWork[DWork[mycounter]++] = u;
    if (ssIsSpecialSampleHit("1")) {
        /* In general, processing on DWork[0:10] can be done
           here, but it does cause the faster rate to have
           nonuniform processing requirements (every 10th hit,
           more code needs to be run).*/
        DWork[10] = sum(DWork[0:9])/10;
        DWork[mycounter] = 0;
    }
}
if (ssIsSampleHit("1")) {
    /* Processing on DWork[10] can be done here before
       outputting. This code runs on every hit of the
       slower task. */
    y = DWork[10];
}
```

Rules for Properly Handling Slow-to-Fast Transitions

When output rates are faster than input rates, input should only be read at the rate that is associated with the input port sample time, observing the following rules:

- Always read input from the update function.
- Use no special sample hit checks when reading input.
- Write the input to a DWork.

- When there is a special sample hit between the rates, copy the DWork into a second DWork in the output function.
- Write the second DWork to the output at every hit of the output sample rate.

The block can request that the input port be made local but it cannot be set to reusable. The output port can be set to local and reusable.

As in the fast-to-slow transition case, the input should not be read by any rate other than the one assigned to the input port. Similarly, the output should not be written to at any 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, only one DWork is needed. 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: $y = u(t_{slow}-1)$
- Input: Set to local, will be local if output/update are combined (ERT) otherwise will be global. Set to not reusable because input needs to be preserved until the update function runs.
- Output: local and reusable
- DirectFeedthrough: no

```
OutputFcn
if (ssIsSampleHit(".1") {
```

```

        if (ssIsSpecialSampleHit("1")) {
            DWork[1] = DWork[0];
        }
        y = DWork[1];
    }
    UpdateFcn
    if (ssIsSampleHit("1")) {
        DWork[0] = u;
    }
}

```

An alternative, optimized approach can be used by some algorithms:

- Input: Set to local, will be local if output/update are combined (ERT) otherwise will be global. Set to not reusable because input needs to be preserved until the update function runs.
- Output: global and not reusable because it needs to persist between special sample hits.
- DirectFeedthrough: no

```

OutputFcn
if (ssIsSampleHit(".1")) {
    if (ssIsSpecialSampleHit("1")) {
        y = DWork;
    }
}
UpdateFcn
if (ssIsSampleHit("1")) {
    DWork = u;
}

```

Example adding a simple algorithm:

- File: `sfun_multirate_modulate.c`, Equation: $y = \sin(tfast) + u(tslow-1)$
- 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

In this section...

- “Introduction” on page 32-90
- “Implicit Build Support” on page 32-91
- “Specifying Additional Source Files for an S-Function” on page 32-91
- “Using TLC Library Functions” on page 32-93
- “Using the `rtwmakecfg.m` API to Customize Generated Makefiles” on page 32-93

Introduction

User-written S-Function blocks provide a powerful way to incorporate legacy and custom code into the Simulink and Real-Time Workshop development environment. In most cases, you should use S-functions to integrate existing code with Real-Time Workshop generated code. Several approaches to writing S-functions are available as discussed in

- “Writing Noninlined S-Functions” on page 32-9
- “Writing Wrapper S-Functions” on page 32-12
- “Writing Fully Inlined S-Functions” on page 32-23
- “Writing Fully Inlined S-Functions with the `mdlRTW` Routine” on page 32-30
- “Writing S-Functions That Support Code Reuse” on page 32-81
- “Writing S-Functions for Multirate Multitasking Environments” on page 32-82

S-functions also provide the most flexible and capable way of including build information for legacy and custom code files in the Real-Time Workshop build process.

This section discusses the different ways of adding S-functions to the Real-Time Workshop build process.

Implicit Build Support

When building models with S-functions, the Real-Time Workshop code generator automatically adds the appropriate 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 directory 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 appropriate tokens.

Details of the implicit build support follow:

- If the file `sfcnname.h` exists in the same directory as the S-function MEX-file (for example, `sfcnname.mexext`), the directory is added to the include path.
- If the file `sfcnname.c` or `sfcnname.cpp` exists in the same directory as the S-function MEX-file, the Real-Time Workshop code generator adds a makefile rule for compiling files from that directory.
- When an S-function is not inlined with a TLC file, the Real-Time Workshop 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 directory.

Specifying 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 `SFunctionsModules` parameter do not support complete source file path specifications. To use the parameter, the Real-Time Workshop software must be able to find the additional source files when executing the makefile. To ensure that the Real-Time Workshop software can locate the additional files, place them in the same directory as the S-function MEX-file. This will enable you to leverage the implicit build support discussed in “Implicit Build Support” on page 32-91.

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 “Using the `rtwmakecfg.m` API to Customize Generated Makefiles” on page 32-93.

Using 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 Real-Time Workshop 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 “Using the `rtwmakecfg.m` API to Customize Generated Makefiles” on page 32-93.

Using the `rtwmakecfg.m` API to Customize Generated Makefiles

- “Overview” on page 32-93
- “Creating the `rtwmakecfg` Function” on page 32-94
- “Modifying the Template Makefile” on page 32-97
- “Precompiling S-Function Libraries” on page 32-99

Overview

Real-Time Workshop TMFs provide tokens that let you add the following items to generated makefiles:

- Source directories
- Include directories
- 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 Real-Time Workshop software associates this file with your S-function based on its directory location. “Creating the `rtwmakecfg` Function” on page 32-94 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 32-97 discusses the required modifications.

After the TLC phase of the build process, when generating a makefile from the TMF, the Real-Time Workshop code generator searches for an `rtwmakecfg.m` file in the directory that contains the S-function component. If it finds the file, the code generator calls the `rtwmakecfg` function.

Creating the `rtwmakecfg` Function

Create the `rtwmakecfg.m` file containing the `rtwmakecfg` function in the same directory 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
<code>makeInfo.includePath</code>	A cell array that specifies additional include directory names, organized as a row vector. The Real-Time Workshop code generator expands the directory names into include instructions in the generated makefile.
<code>makeInfo.sourcePath</code>	A cell array that specifies additional source directory names, organized as a row vector. You must include the directory names of files entered into the S-function modules field on the S-Function Block Parameters dialog box or into the block's SFunctionModules parameter if they are not in the same directory as the S-function. The Real-Time Workshop code generator expands the directory names into make rules in the generated makefile.
<code>makeInfo.sources</code>	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 Real-Time Workshop 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 <code>sourcePath</code> field.
<code>makeInfo.linkLibsObjs</code>	A cell array that specifies additional, fully qualified paths to object or library files against which the Real-Time Workshop generated code should link. The Real-Time Workshop 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 Real-Time Workshop 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 Real-Time Workshop build process.

Field	Description
<code>makeInfo.precompile</code>	A Boolean flag that indicates whether the libraries specified in the <code>rtwmakecfg.m</code> file exist in a specified location (<code>precompile==1</code>) or if the libraries need to be created in the build directory during the Real-Time Workshop build process (<code>precompile==0</code>).
<code>makeInfo.library</code>	A structure array that specifies additional run-time libraries and module objects, organized as a row vector. The Real-Time Workshop 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
<code>makeInfo.library(n).Name</code>	A character array that specifies the name of the library (without an extension).
<code>makeInfo.library(n).Location</code>	A character array that specifies the directory in which the library is located when precompiled. See the description of <code>makeInfo.precompile</code> in the preceding table for more information. A target can use the <code>TargetPreCompLibLocation</code> parameter to override this value. See “Specifying the Location of Precompiled Libraries” on page 27-8 for details.
<code>makeInfo.library(n).Modules</code>	A cell array that specifies the C or C++ source file base names (without an extension) that comprise the library. Do not include the file extension. The makefile appends the appropriate 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 directory: ',pwd]);
makeInfo.includePath = { fullfile(pwd, 'somedir2') };
makeInfo.sourcePath = {fullfile(pwd, 'somedir2'), fullfile(pwd, 'somedir3')};
makeInfo.sources = { 'src1.c', 'src2.cpp' };
makeInfo.linkLibsObjs = { fullfile(pwd, 'somedir3', 'src3.object'),...
                        fullfile(pwd, 'somedir4', 'mylib.library')};
makeInfo.precompile = 1;
makeInfo.library(1).Name      = 'myprecompiledlib';
makeInfo.library(1).Location = fullfile(pwd,'somedir2','lib');
makeInfo.library(1).Modules   = {'srcfile1' 'srcfile2' 'srcfile3' };

```

Note If a path that you specify in the `rtwmakecfg.m` API contains spaces, the Real-Time Workshop 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 “Enabling the Real-Time Workshop Software to Build When Path Names Contain Spaces” on page 18-39.

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 be appropriate for 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 Directory Names to the Makefile Include Path.

The following TMF code example adds directory 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 “Controlling the Location and Naming of Libraries During the Build Process” on page 27-7.

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<

```

Precompiling 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 directory.

By precompiling S-function libraries, you can optimize system builds. Once your precompiled libraries exist, the Real-Time Workshop 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 directory.
- 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 directory.

Use one of the following methods to set the precompiled library directory.

- Set the `TargetPreCompLibLocation` parameter, as explained in “Specifying the Location of Precompiled Libraries” on page 27-8.
- 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 directory for model `my_model` to directory `lib` under the current working directory.

```
set_param(my_model,'TargetPreCompLibLocation', fullfile(pwd,'lib'));
```

Note If you set both the target directory for the precompiled library files and a target library file suffix, the Real-Time Workshop 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. All fields except `rtwmakecfgDirs` are optional.

Field	Description
<code>rtwmakecfgDirs</code>	<p>A cell array of strings that name the directories containing <code>rtwmakecfg</code> files for libraries to be precompiled. The function uses the <code>Name</code> and <code>Location</code> elements of <code>makeInfo.library</code>, as returned by <code>rtwmakecfg</code>, to specify the name and location of the precompiled libraries. If you set the <code>TargetPreCompLibLocation</code> parameter to specify the library directory, that setting overrides the <code>makeInfo.library.Location</code> setting.</p> <p>Note: The specified model must contain blocks that use precompiled libraries specified by the <code>rtwmakecfg</code> files. This is necessary because the TMF-to-makefile conversion generates the library rules only if the libraries are needed.</p>
<code>libSuffix</code>	<p>A string that specifies the suffix, including the file type extension, to be appended to the name of each library (for example, <code>.a</code> or <code>_vc.lib</code>). The string must include a period (<code>.</code>). You must set the suffix with either this field or the <code>TargetLibSuffix</code> parameter. If you specify a suffix with both mechanisms, the <code>TargetLibSuffix</code> setting overrides the setting of this field.</p>
<code>intOnlyBuild</code>	<p>A Boolean flag. When set to true, the flag indicates the libraries are to be optimized such that they are compiled from integer code only. This field applies to ERT targets only.</p>
<code>makeOpts</code>	<p>A string that specifies an option to be included in the <code>rtwMake</code> command line.</p>
<code>addLibs</code>	<p>A cell array of structures that specify libraries to be built that are not specified by an <code>rtwmakecfg</code> function. Each structure must be defined with two fields that are character arrays:</p> <ul style="list-style-type: none"> • <code>libName</code> — the name of the library without a suffix • <code>libLoc</code> — the location for the precompiled library <p>The TMF can specify other libraries and how those libraries are to be built. Use this field if you need to precompile those libraries.</p>

The following commands set up build specification `build_spec`, which indicates that the files to be compiled are in directory `src` under the current working directory.

```
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);
```

Setting Up Generated Code To Interface With Components in the Run-Time Environment

- Chapter 33, “Configuring the Target Environment”
- Chapter 34, “Interfacing With a Real-Time Operating System”
- Chapter 35, “Interacting with Target Application Data Using the C API”
- Chapter 36, “Generating Model Information for Host-Based ASAP2 Data Measurement and Calibration”
- Chapter 37, “Communicating With Code Executing on a Target System Using Simulink External Mode”
- Chapter 38, “Accessing Memory in Generated Code”
- Chapter 39, “Setting Up Runtime Logging to MAT-Files”
- Chapter 40, “Sharing Utility Functions”



Configuring the Target Environment

- “Configuring the Hardware Implementation” on page 33-2
- “Configuring the Target Hardware Environment” on page 33-16

Configuring the Hardware Implementation

When you use Simulink software to create and execute a model, and Real-Time Workshop software to generate code, you may need to consider up to three platforms, often called *hardware targets*:

- MATLAB Host — The platform that runs MathWorks software during application development
- Embedded Target — The platform on which an application will be deployed when it is put into production
- Emulation Target — The platform on which an application under development is tested before deployment

The same platform might serve in two or possibly all three capacities, but they remain conceptually distinct. Often the MATLAB host and the emulation target are the same. The embedded target is usually different from, and less powerful than, the MATLAB host or the emulation 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 Real-Time Workshop software to generate code for deployment on an *embedded* target, you must provide information about the embedded target hardware and the compiler that you will use with it. The Simulink software uses this information to guarantee bit-true agreement for the results of integer and fixed-point operations performed in simulation and in code generated for the embedded target. The Real-Time Workshop code generator uses the information to create code that executes with maximum efficiency.

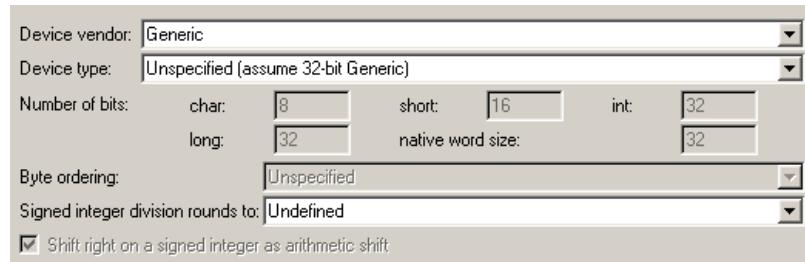
When you generate code for testing on an *emulation* target, you must additionally provide information about the emulation target hardware and the compiler that you will use with it. The code generator uses this information to create code that provides bit-true agreement for the results of integer and fixed-point operations performed in simulation, in code generated for the embedded target, and in code generated for the emulation target. The agreement is guaranteed even though the embedded target and emulation 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.

Describing the Emulation and Embedded Targets

The Configuration Parameters dialog **Hardware Implementation** pane provides options 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:

- **Embedded Hardware** — Describes the embedded target hardware and the compiler that you will use with it. This information affects both simulation and code generation.
- **Emulation Hardware** — Describes the emulation target hardware and the compiler that you will use with it. This information affects only code generation.

The two subpanes provide identical options and value choices. By default, the **Hardware Implementation** subpane initially look like this:



The default assumption is that the embedded target and emulation target are the same, so the **Emulation Hardware** subpane by default does not need to specify anything and contains only a checked option labeled **None**. Code generated under this configuration will be suitable for production use, or for testing in an environment identical to the production environment.

If you clear the check box, the **Emulation Hardware** subpane appears, initially showing the same values as the **Emulation Hardware** subpane. If you change any of these values, then generate code, the code will be able to execute in the environment specified by the **Emulation Hardware** subpane, but will behave as if it were executing in the environment specified by the **Embedded Hardware** subpane. See “Describing Emulation Hardware Characteristics” on page 33-13 for details.

If you have used the **Real-Time Workshop** pane **General** tab 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.

Options with only one possible value cannot be changed. Any option that has more than one possible value provides a list of legal values. If you specify any 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 options do not control hardware or compiler behavior in any way. Their purpose is solely to describe hardware and compiler properties to MATLAB software, which uses the information to generate code that is correct for the platform, 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 options in the **Embedded Hardware** and **Emulation Hardware** subpanes. Subsequent sections describe considerations that apply only to one or the other subpane. For more about **Hardware Implementation** options, see “Hardware Implementation Pane”. To see an example of **Hardware Implementation** pane capabilities, run the `rtwdemo_targetsettings` demo.

Describing the Device Vendor

The **Device vendor** option gives the name of the device vendor. To set the option, 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** option 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 “Registering Additional Device Vendor and Device Type Values” on page 33-5.
-

Describing the Device Type

The **Device type** option selects a hardware device among the supported devices listed for your **Device vendor** selection. To set the option, 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 is appropriate, 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. Options with only one possible value cannot be changed. Any option that has more than one possible value provides a list of legal values. Select values appropriate to your hardware. If the device type is **Custom**, all options can be set, and each option has a list of all possible values.

Registering 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 Real-Time Workshop 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

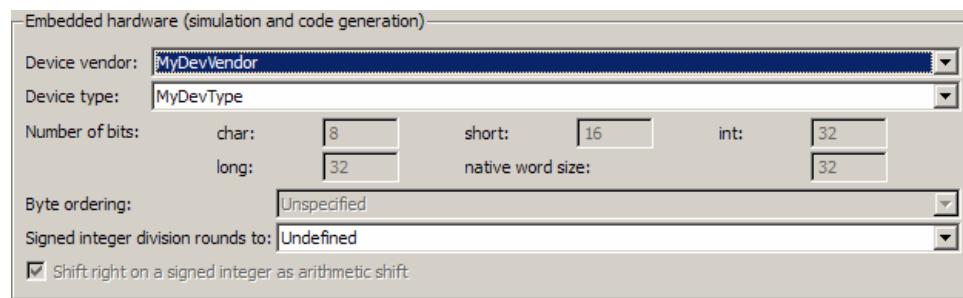
Simulink session. (To register your device information without restarting MATLAB, issue the MATLAB command `sl_refresh_customizations`.)

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);

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]);
    thisDevEndianess = '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 any aliases or legacy names that users might specify that should resolve to this device. Specify each alias or legacy name in the format ' <code>'Vendor->Type'</code> '. (Real-Time Workshop Embedded Coder software provides the utility functions <code>RTW.isHWDeviceTypeEq</code> and <code>RTW.resolveHWDeviceType</code> for detecting and resolving alias values or legacy values when testing user-specified values for the target device type.)

Property	Description
Platform	Cell array of enumerated string values specifying whether this device should be listed in the Embedded hardware subpane (<code>{'Prod'}</code>), the Emulation hardware subpane (<code>{'Target'}</code>), or both (<code>{'Prod', 'Target'}</code>).
setWordSizes	Array of integer sizes to associate with the Number of bits parameters char , short , int , long , and native word size , respectively.
Endianess	String specifying an enumerated value for the Byte ordering parameter: <code>'Unspecified'</code> , <code>'Little'</code> for little Endian, or <code>'Big'</code> for big Endian.
IntDivRoundTo	String specifying an enumerated value for the Signed integer division rounds to parameter: <code>'Zero'</code> , <code>'Floor'</code> , or <code>'Undefined'</code> .
ShiftRightIntArith	Boolean value specifying whether your compiler implements a signed integer right shift as an arithmetic right shift (<code>true</code>) or not (<code>false</code>).
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 <code>'Endianess'</code> , <code>'IntDivRoundTo'</code> , and <code>'ShiftRightIntArith'</code> properties listed above, you can enable individual Number of bits parameters using the property names <code>'BitPerChar'</code> , <code>'BitPerShort'</code> , <code>'BitPerInt'</code> , <code>'BitPerLong'</code> , and <code>'NativeWordSize'</code> .

Describing the Number of Bits

The **Number of bits** options 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**.

- All bit lengths must be multiples of 8, with a maximum of 32.
- The bit length for **long** data must not be less than 32.

Real-Time Workshop 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 Real-Time Workshop integer type names and maps them to the corresponding Simulink names.

Real-Time Workshop Integer Type	Simulink Integer Type
<code>boolean_T</code>	<code>boolean</code>
<code>int8_T</code>	<code>int8</code>
<code>uint8_T</code>	<code>uint8</code>
<code>int16_T</code>	<code>int16</code>
<code>uint16_T</code>	<code>uint16</code>
<code>int32_T</code>	<code>int32</code>
<code>uint32_T</code>	<code>uint32</code>

If no ANSI C type with a matching word size is available, but a larger ANSI C type is available, the Real-Time Workshop 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 any length from 1 (unsigned) or 2 (signed) bits up 32 bits. If the integer length matches the length of an available type, the Real-Time Workshop code generator uses that type. If no matching type is available, the code generator uses the smallest available type that can hold the data, generating code that never uses 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`.

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.

Describing the Byte Ordering

The **Byte ordering** option 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 Real-Time Workshop software generates code that determines the endianness of the target; this is the least efficient option.

Describing Quotient Rounding for Signed Integer Division

ANSI C does not completely define the 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 Real-Time Workshop code generator how the compiler rounds the result of signed integer division. Providing this information does not affect 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 options 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 option 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 options:

N	D	Ideal N/D	Zero	Floor	Undefined
33	4	8.25	8	8	8
-33	4	-8.25	-8	-9	-8 or -9

N	D	Ideal N/D	Zero	Floor	Undefined
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 Real-Time Workshop code generator does not know the signed integer division rounding behavior of the compiler, it must generate fairly costly code in order to guarantee correct results.

The compiler's implementation for signed integer division rounding can be obtained from the compiler documentation, or by experiment if no documentation is available.

Describing Arithmetic Right Shifts on 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** parameter tells the Real-Time Workshop code generator how the compiler implements right shifts on negative integers. Providing this information does not affect 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 twos 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 Real-Time Workshop software generates simple efficient code whenever the Simulink model performs arithmetic shifts on signed integers.
- When the option is cleared, the Real-Time Workshop 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 no documentation is available.

Describing Embedded Hardware Characteristics

"Describing the Emulation and Embedded Targets" on page 33-3 documents the options available on the **Hardware Implementation** subpanes. This section describes considerations that apply only to the **Embedded 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 Real-Time Workshop code generator takes these differences into account when generating code.
- The Simulink product uses some of the information in the **Embedded Hardware** subpane to ensure 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 Real-Time Workshop software generates code that guarantees 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.
- To ensure correctness and efficiency, if you change targets at any point during application development you must reconfigure the hardware implementation parameters for the new target before generating or regenerating code. Bit-true agreement for the results of integer and fixed-point operations in simulation, production code, and test code is *not* guaranteed 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 Lookup Table (n-D) 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 “Rounding” in the *Simulink Fixed Point User’s Guide*.
- When models contain Model blocks, all models that they reference must be configured to use identical hardware settings.

Describing Emulation Hardware Characteristics

“Describing the Emulation and Embedded Targets” on page 33-3 documents the options available on the **Hardware Implementation** subpanes. This section describes considerations that apply only to the **Emulation Hardware** subpane.

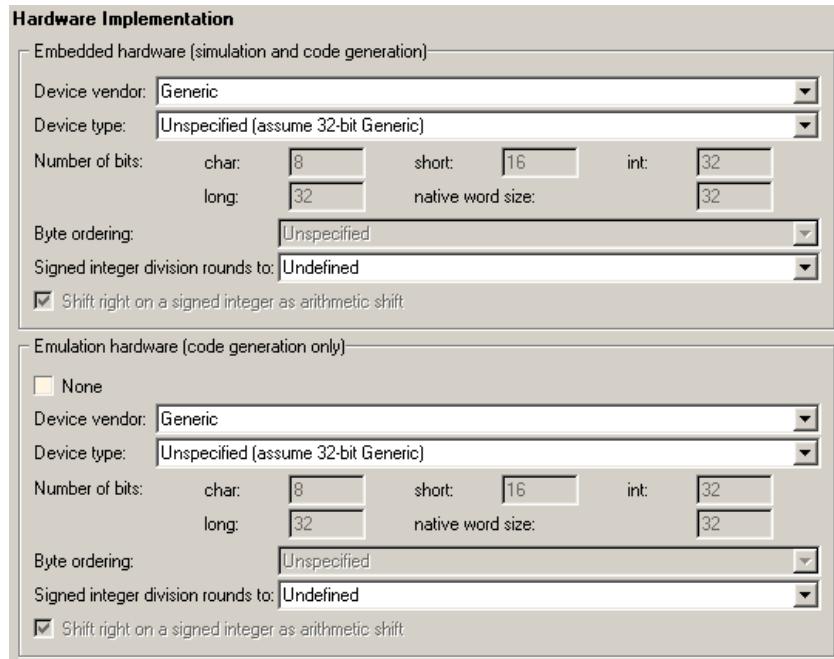
Note (If the **Emulation Hardware** subpane contains a button labeled **Configure current execution hardware device**, see “Updating from Earlier Versions” on page 33-15, then continue from this point.)

The default assumption is that the embedded target and emulation target are the same, so the **Emulation Hardware** subpane by default does not need to specify anything and contains only a selected check box labeled **None**. 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 an emulation target for test purposes, but behaves as if it were running on an embedded target in a production application, you must specify the properties of both targets in the **Hardware Implementation** pane. The **Embedded Hardware** subpane specifies embedded target hardware properties, as described previously. To specify emulation target properties:

- 1 Clear the **None** option in the **Emulation Hardware** subpane.

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



- 2 In the **Emulation Hardware** subpane, specify the properties of the emulation target, using the instructions in “Describing the Emulation and Embedded Targets” on page 33-3

If you have used the **Real-Time Workshop** pane **General** tab 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.

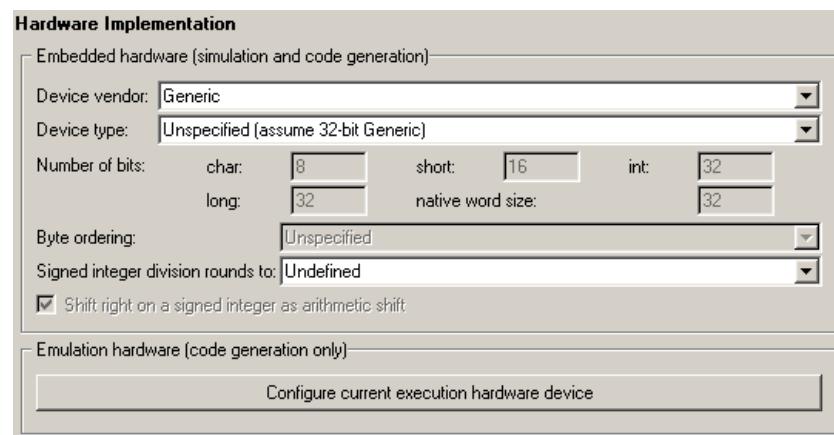
Options with only one possible value cannot be changed. Any option that has more than one possible value provides a list of legal values. If you specify any hardware properties manually, check carefully that their values are consistent with the system target file. Otherwise, the generated code may fail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

If you do not display the **Emulation Hardware** subpane, the Simulink and Real-Time Workshop software defaults every **Emulation Hardware** option to have the same value as the corresponding **Embedded Hardware** option. If you hide the **Emulation 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.

Updating from Earlier Versions

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.

Configuring the Target Hardware Environment

In this section...

[“Specifying Target Interfaces” on page 33-16](#)

[“Selecting and Viewing Target Function Libraries” on page 33-20](#)

Specifying Target Interfaces

Use the **Interface** pane to control which math library is used at run time, whether to include one of three APIs in generated code, and certain other interface options.

To...	Select or Enter...
Specify the target-specific math library to use when generating code	<p>Select C89/C90 (ANSI), C99 (ISO), GNU99 (GNU), or C++ (ISO) for Target function library. (Additional values may be listed if you have created and registered target function libraries with the Real-Time Workshop Embedded Coder product, or if you have licensed any Link or Target products.)</p> <p>Selecting C89/C90 (ANSI) provides the ANSI²⁵ C set of library functions. For example, selecting C89/C90 (ANSI) would result in generated code that calls <code>sin()</code> whether the input argument is double precision or single precision. However, if you select C99 (ISO), the call instead is to the function <code>sinf()</code>, which is single precision. If your compiler supports the ISO²⁶ C math extensions, selecting the ISO C library can result in more efficient code.</p> <p>For more information about target function libraries, see “Selecting and Viewing Target Function Libraries” on page 33-20.</p>
Direct where the Real-Time Workshop code generator should	Select Auto or Shared location for Utility function generation . The shared location directs code for utilities to be placed within

25. ANSI® is a registered trademark of the American National Standards Institute, Inc.

26. ISO® is a registered trademark of the International Organization for Standardization.

To...	Select or Enter...
place fixed-point and other utility code	<p>the <code>slprj</code> directory in your working directory, which is used for building model reference targets. If you select Auto,</p> <ul style="list-style-type: none"> • When the model contains Model blocks, utilities are placed within the <code>slprj/target/_sharedutils</code> directory. • When the model does not contain Model blocks, utilities are placed in the build directory (generally, in <code>model.c</code> or <code>model.cpp</code>).
Specify a string to be added to the variable names used when logging data to MAT-files, to distinguish logging data from Real-Time Workshop and Simulink applications	Enter a prefix or suffix, such as <code>rt_</code> or <code>(_rt</code> , for MAT-file variable name modifier . The Real-Time Workshop 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 Chapter 39, “Setting Up Runtime Logging to MAT-Files” for information on MAT-file data logging.
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 and External mode are mutually exclusive. However, this is not the case for C API and ASAP2 . For more information on working with these interfaces, see Chapter 35, “Interacting with Target Application Data Using the C API”, Chapter 37, “Communicating With Code Executing on a Target System Using Simulink External Mode”, and Chapter 36, “Generating Model Information for Host-Based ASAP2 Data Measurement and Calibration”.

Note Before setting **Target function 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 **C99 (ISO)** and your compiler does not support the ISO C math extensions, compile-time errors likely will occur.

When the Real-Time Workshop Embedded Coder product is installed on your system, the **Interface** pane expands to include several additional options. For details, see “Configuring the Target Hardware Environment” in the Real-Time Workshop Embedded Coder documentation.

For a summary of option dependencies, see “Interface Dependencies” on page 33-18.

For descriptions of Interface pane parameters, see “Real-Time Workshop Pane: Interface” in the Real-Time Workshop reference documentation.

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
Target function library	No	
Utility function generation	Yes	
Support: floating-point numbers (ERT targets only)	No	
Support: non-finite numbers (ERT targets only)	Yes	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
GRT compatible call interface (ERT targets only)	Yes	Requires that you enable Support floating-point numbers and disable Single output/update function
Single output/update function (ERT targets only)	Yes	Disable for GRT compatible call interface

Parameter	Dependencies?	Dependency Details
Terminate function required (ERT targets only)	Yes	
Generate reusable code (ERT targets only)	Yes	
Reusable code error diagnostic (ERT targets only)	Yes	Enabled by Generate reusable code
Pass root-level I/O as (ERT targets only)	Yes	Enabled by Generate reusable code
MAT-file logging	Yes	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 (ERT targets only)	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

Selecting and Viewing Target Function Libraries

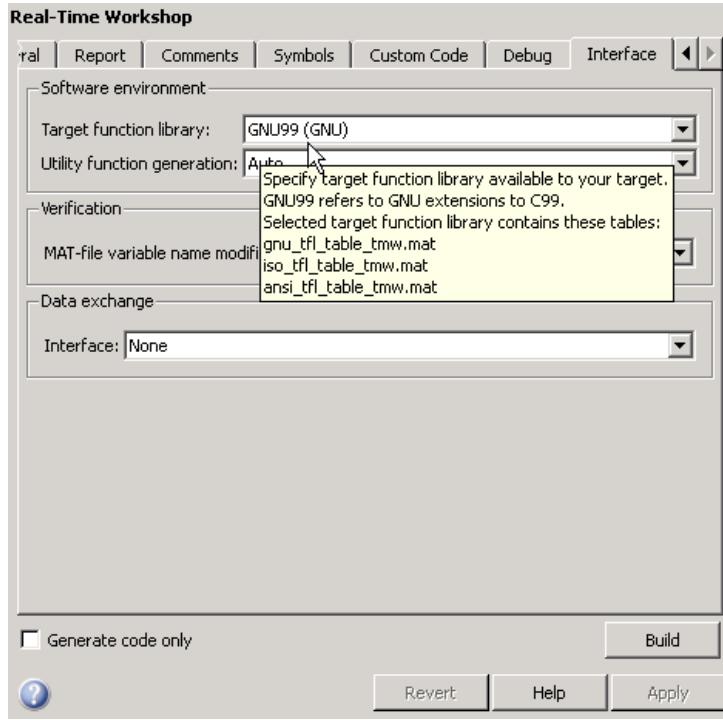
- “Selecting a Target-Specific Math Library for Your Model” on page 33-20
- “Function Replacement Table Overview” on page 33-21
- “Using the Target Function Library Viewer” on page 33-23

Selecting a Target-Specific Math Library for Your Model

A *target function library* (TFL) is a set of one or more function replacement tables that define the target-specific implementations of math functions and operators to be used in generating code for your Simulink model. The Real-Time Workshop product provides default TFLs, which you can select from the **Target function library** drop-down list on the **Interface** pane of the Configuration Parameters dialog box.

TFL	Description	Contains tables...
C89/C90 (ANSI)	Generates calls to the ISO/IEC 9899:1990 C standard math library for floating-point functions.	ansi_tfl_table_tmw.mat
C99 (ISO)	Generates calls to the ISO/IEC 9899:1999 C standard math library.	iso_tfl_table_tmw.mat ansi_tfl_table_tmw.mat
GNU99 (GNU)	Generates calls to the Free Software Foundation's GNU gcc math library, which provides C99 extensions as defined by compiler option <code>-std=gnu99</code> .	gnu_tfl_table_tmw.mat iso_tfl_table_tmw.mat ansi_tfl_table_tmw.mat
C++ (ISO)	Generates calls to the ISO/IEC 14882:2003 C++ standard math library.	iso_cpp_tfl_table_tmw.mat private_iso_cpp_tfl_table_tmw.mat iso_tfl_table_tmw.mat ansi_tfl_table_tmw.mat

TFL tables provide the basis for replacing default math functions and operators in your model code with target-specific code. If you select a library and then hover over the selected library with the cursor, a tool tip is displayed that describes the TFL and lists the function replacement tables it contains. Tables are listed in the order in which they are searched for a function or operator match.



The Real-Time Workshop product allows you to view the content of TFL function replacement tables using the Target Function Library Viewer, as described in “Using the Target Function Library Viewer” on page 33-23. If you are licensed to use the Real-Time Workshop Embedded Coder product, you additionally can create and register the function replacement tables that make up a TFL.

Function Replacement Table Overview

Each TFL function replacement table contains one or more table entries, with each table entry representing a potential replacement for a single math function or an operator. Each table entry provides a mapping between a *conceptual view* of the function or operator (similar to the Simulink block view of the function or operator) and a *target-specific implementation* of that function or operator.

The conceptual view of a function or operator is represented in a TFL table entry by the following elements, which identify the function or operator entry to the code generation process:

- A function or operator key (a function name such as 'cos' or an operator ID string such as 'RTW_OP_ADD')
- A set of conceptual arguments that observe Simulink naming ('y1', 'u1', 'u2', ...), along with their I/O types (output or input) and data types
- Other attributes, such as fixed-point saturation and rounding characteristics for operators, as needed to identify the function or operator to the code generation process as exactly as required for matching purposes

The target-specific implementation of a function or operator is represented in a TFL table entry by the following elements:

- The name of an implementation function (such as 'cos_dbl' or 'u8_add_u8_u8')
- A set of implementation arguments, along with their I/O types (output or input) and data types
- Parameters providing the build information for the implementation function, including header file and source file names and paths as necessary

Additionally, a TFL table entry includes a priority value (0-100, with 0 as the highest priority), which defines the entry's priority relative to other entries in the table.

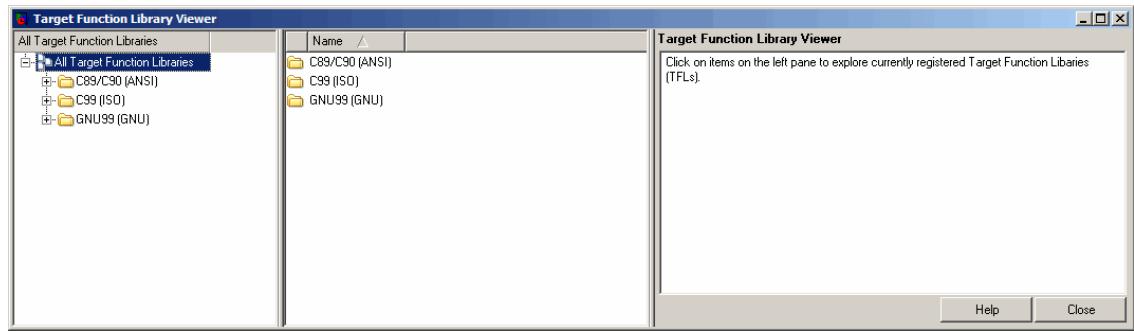
During code generation for your model, when the code generation process encounters a call site for a math function or operator, it creates and partially populates a TFL entry object, for the purpose of querying the TFL database for a replacement function. The information provided for the TFL query includes the function or operator key and the conceptual argument list. The TFL entry object is then passed to the TFL and, if there is a matching table entry in the TFL, a fully-populated TFL entry, including the implementation function name, argument list, and build information, is returned to the call site and used to generate code.

Within the TFL that is selected for your model, the tables that comprise the TFL are searched in the order in which they are listed (in the left or right pane of the TFL Viewer or in the TFL's **Target function library** tool tip). Within each table, if multiple matches are found for a TFL entry object, priority level determines the match that is returned. A higher-priority (lower-numbered) entry will be used over a similar entry with a lower priority (higher number).

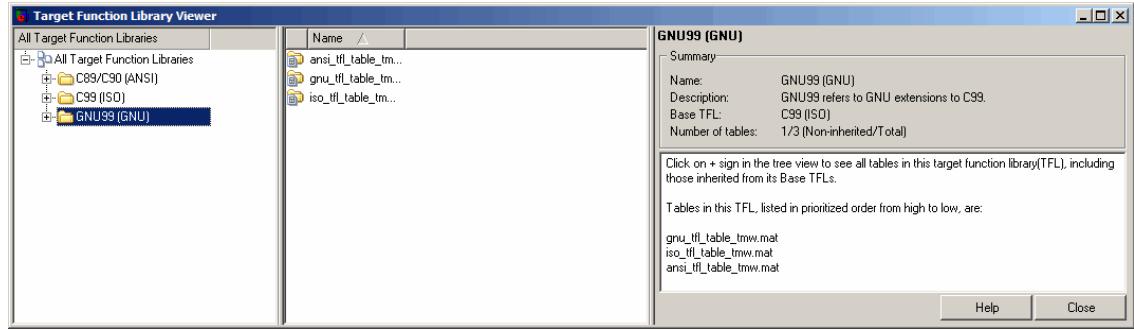
Using the Target Function Library Viewer

The Target Function Library Viewer allows you to examine the content of TFL function replacement tables. (For an overview of function replacement tables and the information they contain, see the preceding section.) To launch the Viewer with all currently registered TFLs displayed, issue the following MATLAB command:

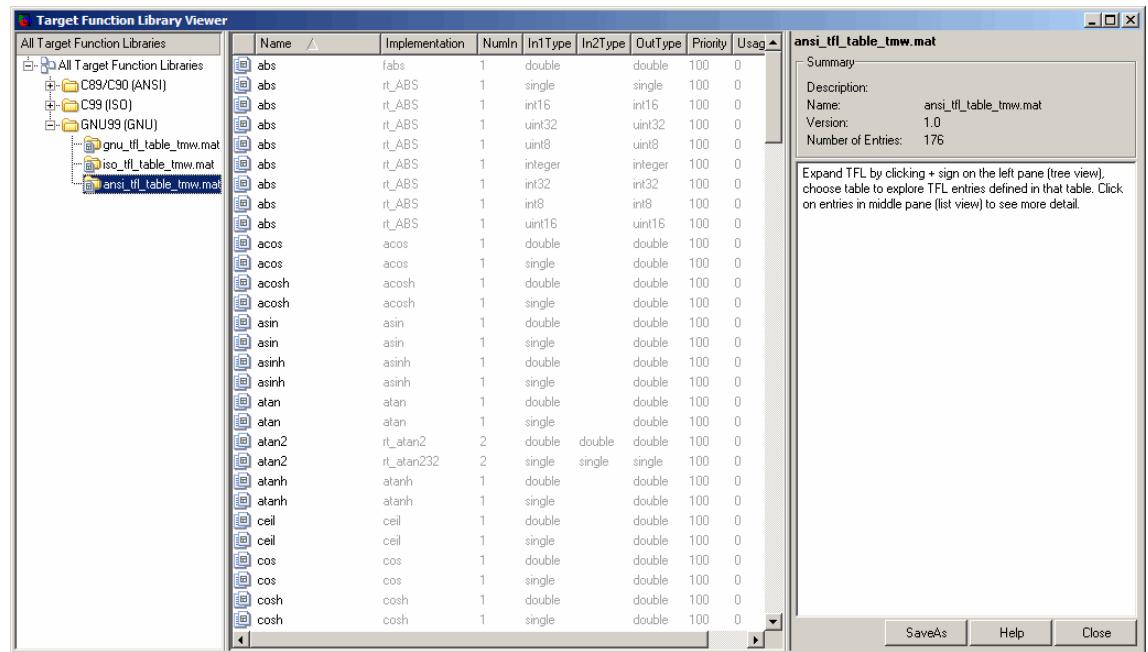
```
>> RTW.viewTfl
```



Select the name of a TFL in the left pane, and the Viewer displays information about the TFL in the right pane. For example, the tables that make up the TFL are listed in priority order. In the following display, the GNU TFL has been selected.



Click the plus sign (+) next to a TFL name in the left pane to expand its list of tables, and select a table from the list. The Viewer displays all function and operator entries in the selected table in the middle pane, along with abbreviated table entry information for each entry. In the following display, the ANSI table has been selected.

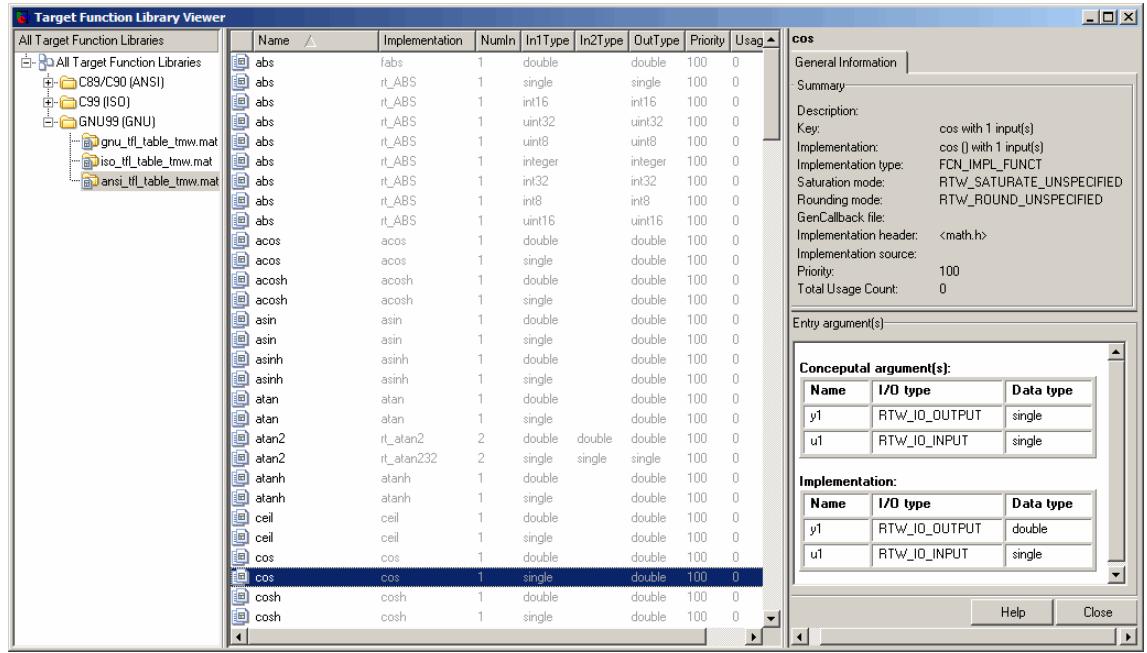


The following fields appear in the abbreviated table entry information provided in the middle pane:

Field	Description
Name	Name of the function or ID of the operator to be replaced (for example, cos or RTW_OP_ADD).
Implementation	Name of the implementation function, which can match or differ from Name.
NumIn	Number of input arguments.
In1Type	Data type of the first conceptual input argument.
In2Type	Data type of the second conceptual input argument.
OutType	Data type of the conceptual output argument.

Field	Description
Priority	The entry's search priority, 0-100, relative to other entries of the same name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. The default is 100. If the table provides two implementations for a function or operator, the implementation with the higher priority will shadow the one with the lower priority.
UsageCount	Not used.

Select a function or operator entry in the middle pane. The Viewer displays detailed information from the table entry in the right pane. In the following display, the second entry for the cos function has been selected.

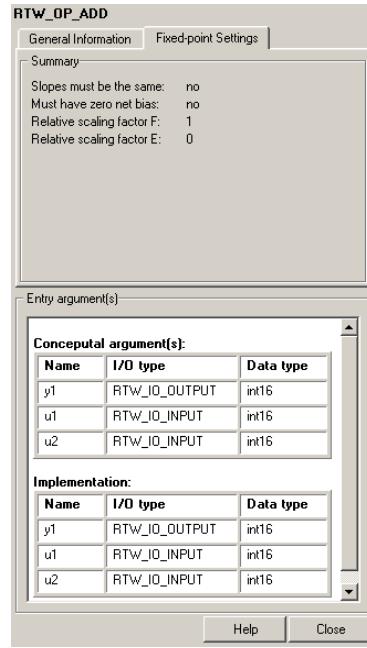


The following fields appear in the detailed table entry information provided in the right pane.

Field	Description
Description	Text description of the table entry (can be empty).
Key	Name of the function or ID of the operator to be replaced (for example, <code>cos</code> or <code>RTW_OP_ADD</code>), and the number of conceptual input arguments.
Implementation	Name of the implementation function, and the number of implementation input arguments.
Implementation type	Type of implementation: <code>FCN_IMPL_FUNCT</code> for function or <code>FCN_IMPL_MACRO</code> for macro.
Saturation mode	Saturation mode supported by the implementation function for an operator replacement: <code>RTW_SATURATE_ON_OVERFLOW</code> , <code>RTW_WRAP_ON_OVERFLOW</code> , or <code>RTW_SATURATE_UNSPECIFIED</code> .
Rounding mode	Rounding mode supported by the implementation function for an operator replacement: <code>RTW_ROUND_FLOOR</code> , <code>RTW_ROUND_CEILING</code> , <code>RTW_ROUND_ZERO</code> , <code>RTW_ROUND_NEAREST</code> , <code>RTW_ROUND_NEAREST_ML</code> , <code>RTW_ROUND_SIMPLEST</code> , <code>RTW_ROUND_CONV</code> , or <code>RTW_ROUND_UNSPECIFIED</code> .
GenCallback file	Not used.
Implementation header	Name of the header file that declares the implementation function.
Implementation source	Name of the implementation source file.
Priority	The entry's search priority, 0-100, relative to other entries of the same name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. The default is 100. If the table provides two implementations for a function or operator, the implementation with the higher priority will shadow the one with the lower priority.

Field	Description
Total Usage Count	Not used.
Conceptual argument(s)	Name, I/O type (RTW_IO_OUTPUT or RTW_IO_INPUT), and data type for each conceptual argument.
Implementation	Name, I/O type (RTW_IO_OUTPUT or RTW_IO_INPUT), and data type for each implementation argument.

If you select an operator entry, an additional tab containing fixed-point setting information is displayed in the right pane. For example:



The following fields appear in the fixed-point setting information provided in the right pane:

Field	Description
Slopes must be the same	Indicates whether TFL replacement request processing must check that the slopes on all arguments (input and output) are equal. Used with fixed-point addition and subtraction replacement to disregard specific slope and bias values and map relative slope and bias values to a replacement function.
Must have zero net bias	Indicates whether TFL replacement request processing must check that the net bias on all arguments is zero. Used with fixed-point addition and subtraction replacement to disregard specific slope and bias values and map relative slope and bias values to a replacement function.
Relative scaling factor F	Slope adjustment factor (F) part of the relative scaling factor, $F2^E$, for relative scaling TFL entries. Used with fixed-point multiplication and division replacement to map a range of slope and bias values to a replacement function.
Relative scaling factor E	Fixed exponent (E) part of the relative scaling factor, $F2^E$, for relative scaling TFL entries. Used with fixed-point multiplication and division replacement to map a range of slope and bias values to a replacement function.

Interfacing With a Real-Time Operating System

- “Resources for Wind River Systems Tornado Applications” on page 34-2
- “Wind River Systems Tornado Application Architecture” on page 34-4
- “Installing the Wind River Systems Tornado Software” on page 34-10
- “Implementing a Wind River Systems Tornado Application” on page 34-12
- “Building the Application” on page 34-22
- “Automatic Download and Execution” on page 34-22
- “Manual Download and Execution” on page 34-24

Resources for Wind River Systems Tornado Applications

In this section...

“Introduction” on page 34-2

“The Tornado Environment” on page 34-2

“The Tornado Target” on page 34-3

“Block Library for Wind River Systems VxWorks” on page 34-3

Introduction

This section describes resources provided by The MathWorks and Wind River Systems, Inc., for converting a Simulink model to a real-time VxWorks application.

The Tornado Environment

The Tornado environment is an integrated development and real-time execution environment that is available from Wind River Systems, Inc. You can use the Tornado environment to develop an application on a host system, download the application to a real-time processor, and run the application, optionally monitoring its activities and changing its parameters. The Tornado environment has many features similar to The Open Group UNIX platforms and runs on a variety of host systems and real-time processors. The Tornado environment includes

- **Application building tools:** Compiler, Linker, Make utility, Archive utility
- **Interactive development tools:** Editor, Debugger, Browser, Configuration tool, Command shell
- **Wind River Systems VxWorks:** A high-performance real-time operating system
- **StethoScope:** An optional data acquisition and graphical monitoring tool

The *Tornado User’s Guide*, and other Tornado documentation give complete information about the Tornado environment. This chapter does not repeat information that is available in the Tornado documentation.

The Tornado Target

To support the Tornado environment, the Real-Time Workshop product provides the *Tornado target*, which you can use to convert a Simulink model to code that works with Tornado tools, runs under the VxWorks real-time operation system (RTOS), and can be monitored by Tornado StethoScope, or monitored and tuned with Simulink external mode. See Chapter 37, “Communicating With Code Executing on a Target System Using Simulink External Mode” for information about Simulink external mode.

Block Library for Wind River Systems VxWorks

The *VxWorks block library* (`vxlib1`) is part of the Real-Time Workshop library. You can access the VxWorks block library by opening the Simulink Library Browser, expanding the **Real-Time Workshop** entry, and clicking **VxWorks**. Alternatively, type `vxlib1` at the MATLAB prompt.

The blocks in the VxWorks block library allow you to model and generate code for asynchronous event handling, including servicing hardware-generated interrupts, maintenance of timers, asynchronous read and write operations, and spawning asynchronous tasks under the VxWorks RTOS. See “Handling Asynchronous Events” on page 5-34 for a detailed description of the blocks in the VxWorks block library.

Note The older Interrupt Templates library (`vxlib`) is obsolete. It is provided only to allow models created prior to Real-Time Workshop version 6.0 to continue to operate. If you have models that use `vxlib` blocks, you should change them to use `vxlib1` blocks.

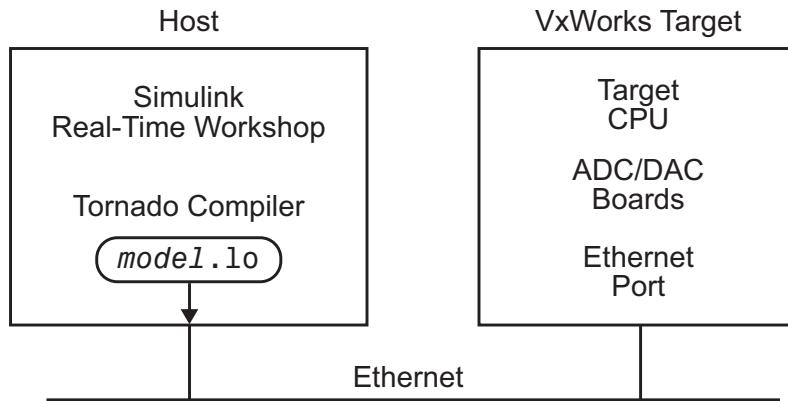
Wind River Systems Tornado Application Architecture

In this section...

- “Hardware Architecture” on page 34-4
- “Software Architecture” on page 34-5
- “Module Architecture” on page 34-6
- “Host Processes” on page 34-6
- “Wind River Systems VxWorks Tasks” on page 34-7

Hardware Architecture

In a typical real-time system based on the VxWorks RTOS, the hardware consists of a PC or The Open Group UNIX host running the Simulink and Real-Time Workshop products, connected by Ethernet to a VxWorks CPU. The chassis containing the VxWorks CPU may also contain I/O boards with A/D and D/A converters to communicate with external hardware. The next figure shows the arrangement.



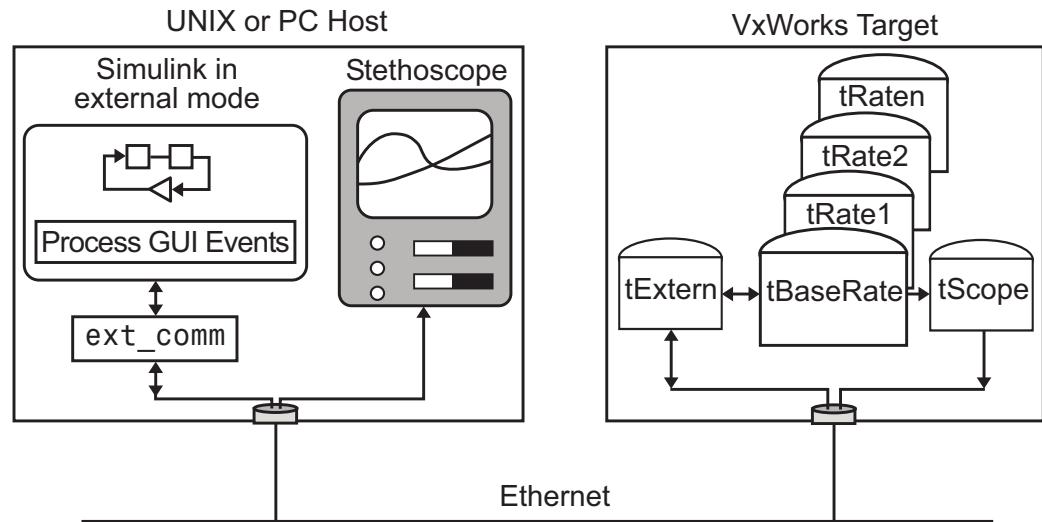
In this figure and the following sections, the VxWorks CPU and associated I/O boards are called the *VxWorks target*. Be careful not to confuse the VxWorks target with the Tornado target. The former is hardware; the latter is a software specification that the Real-Time Workshop product uses to convert a Simulink model to a VxWorks application.

Software Architecture

The real-time code is compiled on the UNIX or PC host using the cross-compiler supplied with the VxWorks package. The object file (*model.1o*) output from the Real-Time Workshop program builder is downloaded to the VxWorks target over the Ethernet connection using the Tornado WindSh command.

The real-time program executes on the VxWorks target, and interfaces with external hardware using the I/O devices installed on the target. If you have configured the program appropriately, you can monitor it using StethoScope, or monitor it and change its parameters using Simulink external mode. StethoScope and the Simulink environment execute on the host workstation, and use tasks on the VxWorks target to handle communication.

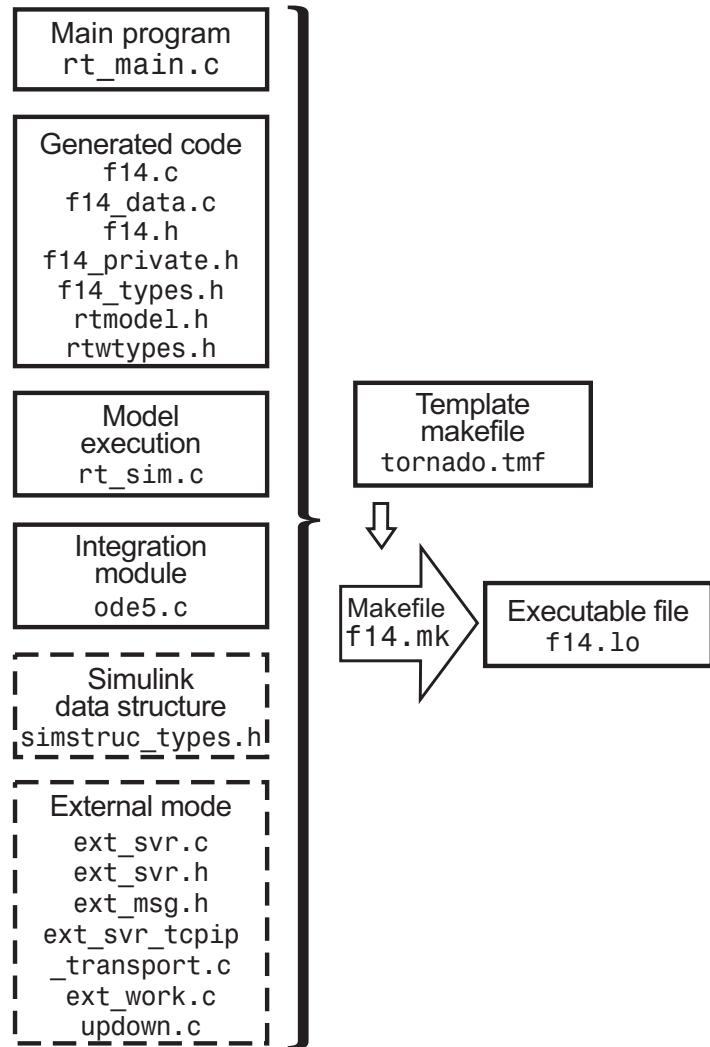
This figure shows the structure of a VxWorks application. The figure shows both StethoScope and the Simulink environment, but only one of them can be used with a given compiled image. To switch to the other interface, you must rebuild with appropriate options, then redownload and restart the program. See “Setting Target-Specific Options” on page 34-15 for more information.



The real-time program creates VxWorks tasks to run on the real-time system, and to communicate with the Simulink engine if the program uses Simulink external mode. StethoScope creates its own tasks to collect data.

Module Architecture

This figure shows the code modules used to build a VxWorks real-time program for the f14 example model. Dashed boxes indicate optional modules, such as those that implement external mode.



Host Processes

Various processes can run on the host side to implement Tornado tools and Simulink capabilities, and optionally to communicate with the real-time

program over the Internet. The host processes that communicate with the real-time program depend on whether you use StethoScope or Simulink external mode.

- If you use StethoScope, the StethoScope user interface module runs on the host. This module communicates with the real-time program running on the VxWorks target to retrieve model data and plot time histories.
- If you use Simulink external mode, and change a parameter in the block diagram, the Simulink engine calls the external link MEX-file to download any new parameter values to the real-time program.

Wind River Systems VxWorks Tasks

Various tasks run on the VxWorks target. Some execute the real-time program, while others are optional and provide communication with the host.

Execution Tasks

You can run the real-time program in either single-tasking or multitasking mode. The code for both modes is located in

matlabroot/rtw/c/tornado/rt_main.c

The Real-Time Workshop product compiles and links *rt_main.c* with the model code during the build process. When the real-time program begins execution, *rt_main.c* starts tasks for single-tasking or multitasking if they are not already running.

Single-Tasking. By default, the model is run as one task, *tSingleRate*. This might actually provide the best performance (highest base sample rate) depending on the model.

The *tSingleRate* task runs at the base rate of the model and executes all necessary code for the slower sample rates. Execution of the *tSingleRate* task is normally blocked by a call to the VxWorks *semTake* routine. When a clock interrupt occurs, the interrupt service routine calls the *semGive* routine, which causes the *semTake* call to return. Once enabled, the *tSingleRate* task executes the model code for one time step. The loop then waits at the top by again calling *semTake*.

For more information about the `semTake` and `semGive` routines, refer to the *VxWorks Reference Manual*. By default, it runs at a relatively high priority (30), which allows it to execute without interruption from background system activity.

Multitasking. Optionally, the model can run as multiple tasks, one for each sample rate in the model.

- `tBaseRate` — This task executes the components of the model code run at the base (highest) sample rate. By default, it runs at a relatively high priority (30), which allows it to execute without interruption from background system activity.
- `tRaten` — The program also spawns a separate task for each additional sample rate in the system. These additional tasks are named `tRate1`, `tRate2`, ..., `tRaten`, where n is the slowest sample rate in the system. The priority of each additional task is one lower than its predecessor (`tRate1` has a lower priority than `tBaseRate`).

Communication Tasks

You can optionally monitor and change the parameters of the executing real-time program using either StethoScope or Simulink external mode, but not both with the same compiled image.

StethoScope Tasks. To enable real-time data collection and display, StethoScope runs one or more VxWorks tasks, such as `tScopeDaemon` and `tScopeLink`. In single-tasking mode, `tSingleRate` collects signals; in multitasking mode, `tBaseRate` collects them. Both perform the collection on every base time step. When idle time exists because the model is waiting for the next time step, a StethoScope task sends the data to the host for display. For more about StethoScope tasks, see the VxWorks documentation.

External Mode Tasks. To enable changing model parameters and collecting data for display, Simulink external mode runs the VxWorks task `tExtern`. The source code for `tExtern` is located in `matlabroot/rtw/c/src/ext_svr.c`. The `tExtern` task implements the server side of a socket stream connection that accepts data transferred between the Simulink engine and the real-time program. The task runs at a lower priority than the lowest priority model task. To tune parameters, `tExtern` waits for a message to arrive from the Simulink engine. When a message arrives, `tExtern` retrieves it and modifies the specified parameters accordingly.

Installing the Wind River Systems Tornado Software

In this section...

- “Introduction” on page 34-10
- “Installing and Configuring the Tornado Environment” on page 34-10
- “Connecting to the Wind River Systems VxWorks Target” on page 34-10
- “Verifying the Tornado Installation” on page 34-11

Introduction

Before you can use the Tornado environment with the Simulink and Real-Time Workshop products to create VxWorks applications, you must perform some installation, configuration, and verification steps. See the Tornado documentation for instructions on all aspects of Tornado installation and configuration. This section provides additional information specific to the Simulink and Real-Time Workshop products.

Installing and Configuring the Tornado Environment

The first step is to install and configure the Tornado environment on your host and target hardware. If you want to use Simulink external mode with your VxWorks program, you must configure your VxWorks kernel to support sockets by including the INCLUDE_NET_INIT, INCLUDE_NET_SHOW, and INCLUDE_NETWORK options in your VxWorks image.

Connecting to the Wind River Systems VxWorks Target

After completing the installation process, you are ready to connect the host workstation to the VxWorks target. The first step is ensuring that the Tornado registry (wtxregd) is running. Typically, it is configured to start running by the host operating system automatically at reboot. However, it can also be started interactively.

Next, start the target server. This server provides communication between the Tornado tools on the host and the agent on the VxWorks target. You can

do this from within the Tornado development environment. On Microsoft Windows, you can use a command shell to enter:

```
tgtsvr target_network_name -A -V
```

Additional options might be required, such as `-c` for the VxWorks core image location.

You may need to enter a routing table entry into the VxWorks system if your host is not on the same local network (subnet) as the VxWorks system. See `routeAdd` in the *VxWorks Reference Guide* for more information.

To check that the VxWorks system can properly respond to your host over the network, use the host command:

```
ping <target_name>
```

Verifying the Tornado Installation

To verify the installation, run one of the VxWorks demonstration programs to ensure you can boot your VxWorks target and download object files to it. Consult the Tornado documentation as needed to configure and debug your Tornado installation. Be sure that it is fully functional before you try to use it with the Simulink and Real-Time Workshop products.

Implementing a Wind River Systems Tornado Application

In this section...

- “Designing the Simulink Model” on page 34-12
- “Adding Device Driver Blocks” on page 34-12
- “Specifying Code Generation Options” on page 34-13
- “Configuring the Template Makefile” on page 34-19

Designing the Simulink Model

A Simulink model to be converted into a Wind River Systems VxWorks real-time program can implement any Simulink capability supported by the Real-Time Workshop product, and has no constraints beyond those that apply to any model intended for Real-Time Workshop code generation. See the “Simulink” documentation and “Real-Time Workshop” documentation for information.

Adding Device Driver Blocks

The real-time program communicates with the I/O devices installed in the VxWorks target chassis by using a set of device drivers. These device drivers contain the necessary code that runs on the target processor for interfacing to specific I/O devices.

To make device drivers easy to use, they are implemented as Simulink S-functions using C/C++ MEX-files. This means you can connect them to your model like any other block and the code generator automatically includes a call to the block’s C/C++ code in the generated code. You can have multiple instances of device driver blocks in your model.

You can inline S-functions by using the Target Language Compiler. Inlining allows you to restrict function calls to only those that are necessary for the S-function. This can greatly increase the efficiency of the S-function. For more information about inlining S-functions, see “Creating Component Object Libraries and Enhancing Simulation Performance” on page 3-92 and the *Real-Time Workshop Target Language Compiler* documentation.

For additional information on creating device drivers, see the Real-Time Workshop Embedded Coder documentation and “Creating Component Object Libraries and Enhancing Simulation Performance” on page 3-92.

Specifying Code Generation Options

To specify the code generation options necessary for a Tornado application, open the Model Explorer or the Configuration Parameters dialog box, then select configuration panes and set options as described in this section. The options that you specify set the appropriate macros in the template makefile, causing any necessary additional steps to be performed during the build process.

Depending on your application, you may also want to specify other code generation options, such as those that control parameter inlining and tunability. You must also specify some information by editing the template makefile, as described in “Configuring the Template Makefile” on page 34-19.

Selecting the Tornado Target

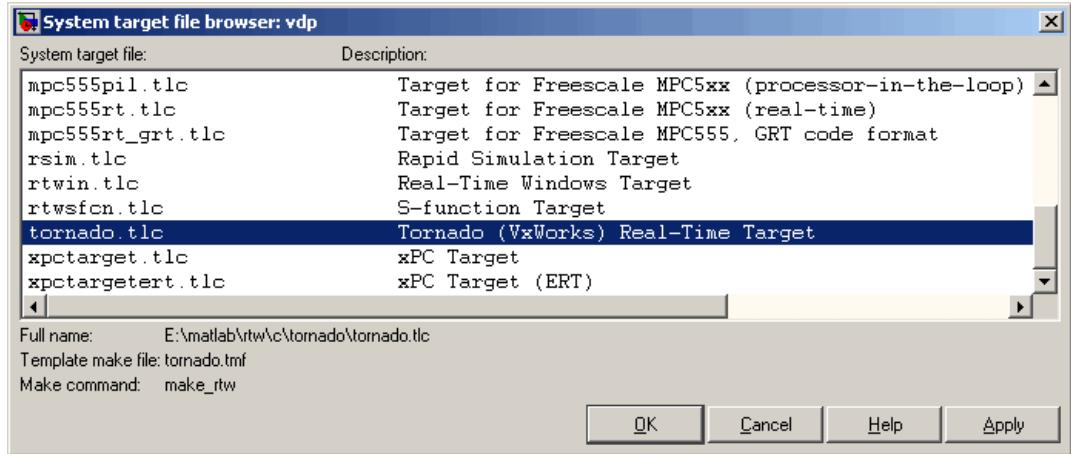
To specify the Tornado (VxWorks) target for code generation,

1 Select the **Real-Time Workshop > General** pane.

2 Click **Browse**.

The System Target File Browser opens.

3 Select `tornado.tlc` (the Tornado target) in the left column, then click **OK**.



Specifying the Tornado target adds a **Real-Time Workshop > Tornado Target** pane and sets **Real-Time Workshop > General** options as follows:

- **System target file:** `tornado.tlc`
- **TLC Options:** `-p0 -aWarnNonSaturatedBlocks=0`
- **Make command:** `make_rtw`
- **Template makefile:** `tornado.tmf`

If you will use a template makefile with a nondefault name, as described in “Configuring the Template Makefile” on page 34-19, you must replace the default **Template makefile** name `tornado.tmf` with the name you want to use instead.

Specifying the Solver

Because generated code cannot use variable-step solvers as a Simulink model can, you must specify a fixed-step solver and set other options:

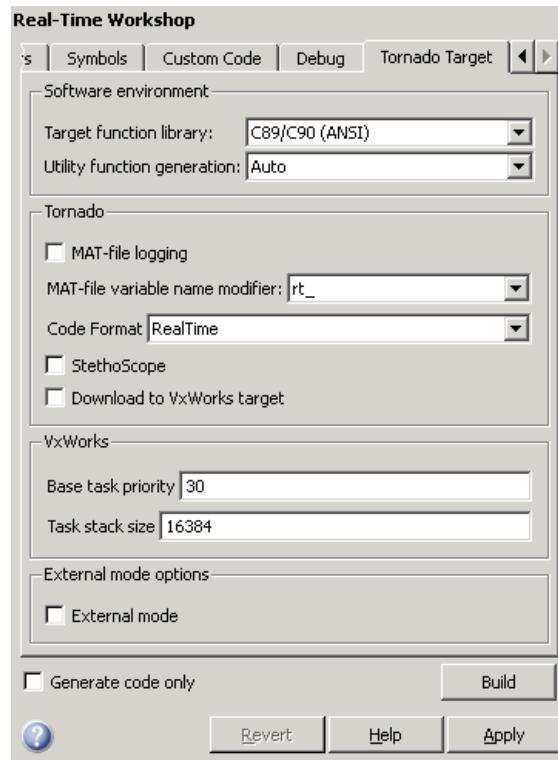
- 1 Select the **Solver** pane.
- 2 Set **Type** to **Fixed-step**.
- 3 Select a **Solver** integration algorithm as appropriate to your model:
 - For a model with any continuous states, choose a fixed-step algorithm.

- For a discrete model, choose **Discrete** (no continuous states).

4 Set **Fixed-step size** to the desired integration step size.

Setting Target-Specific Options

Once you have selected the Tornado target, as described above, you can set target-specific code generation options. First select the **Tornado Target** pane. The next figure shows the default Tornado configuration.



You can change options on the **Tornado Target** pane to enable and configure the capabilities described in this section.

MAT-File Logging. Select **MAT-file logging** to enable data logging during program execution. The program creates a file named *model.mat* at the end of program execution. This file contains the variables that you specified in the **Configuration Parameters > Data Import/Export** pane. See “Data Import/Export Pane” and “MAT-file logging”.

The Real-Time Workshop code generator adds a prefix or suffix to the names of the **Data Import/Export** pane variables that you select for logging. The **MAT-file variable name modifier** menu lets you select this prefix or suffix. See “Data Import/Export Pane” and “MAT-file variable name modifier”.

By default, the MAT-file is created in the root directory of the VxWorks current default device. This is typically the host file system from which the VxWorks system was booted. Other remote file systems can be used as a destination for the MAT-file using rsh or ftp network devices or NFS. See the *VxWorks Programmer’s Guide* for more information.

If a device or filename other than the default is desired, use the OPTS flag to add “-DSAVEFILE=filename” (with quotes) to the **Make command** shown on the **Real-Time Workshop > General** pane. For example:

```
make_rtw OPTS="-DSAVEFILE=MyFileName"
```

Generated Code Format. Use **Code format** to specify the **RealTime** or **RealTimeMalloc** code generation format.

StethoScope Monitoring. Select **StethoScope** to enable the use of StethoScope with the generated executable. You cannot select both **StethoScope** and **External mode** at the same time.

With StethoScope, you can access the output of any block in the model (as executed by the real-time program) and display this data on the host. You can install signals in StethoScope in one of two ways:

- From the real-time program using the **rtBlockIOSignals** data structure. See Chapter 35, “Interacting with Target Application Data Using the C API” for details.
- With the Tornado **WindSh** command while the real-time program is running. See the *StethoScope User’s Manual* for details.

When you start the downloaded program, you must specify two command-line arguments that control the block names used by StethoScope. See “Starting the Program” on page 34-25 for information about these and other command-line arguments.

Automatic Download. Select **Download to VxWorks target** to enable automatic downloading of the generated program.

Base Task Priority. Specify the priority with which the base rate task for the model is to be spawned. The default value is 30. For a multirate, multitasking model, the Real-Time Workshop product increments the priority of each substrate task by one. For example, if a model includes two substrate tasks and the priority of the base rate task is 30, the priorities for the substrate tasks are 31 and 32.

Note, the value you specify for this option will be overridden by a base priority specified in a call to the `rt_main()` function spawned as a task.

Task Stack Size. Specify the stack size to be used in calls to the VxWorks function `taskSpawn()`, which spawns tasks for the base rate and substrates of the model. The default size is 16384 bytes.

The VxWorks system target file uses the value you specify for this option to set the TLC variables `MaxStackSize` and `MaxStackVariableSize`. The variables are set as follows:

<code>MaxStackSize</code>	One-half the size of the Task stack size setting (8192 bytes by default)
<code>MaxStackVariableSize</code>	One-half the size of the <code>MaxStackSize</code> (4096 bytes by default)

These variable settings help ensure the generated code does not exceed the stack size allocated for each task. Thus, you should tune this option to the match the requirements of your generated code.

Simulink External Mode. Simulink external mode provides a mechanism to monitor signals in your model and download new parameter values to the executing real-time program. See Chapter 37, “Communicating With Code Executing on a Target System Using Simulink External Mode” for information about external mode. This section does not repeat the information available there.

Communication between the Simulink engine and a real-time VxWorks program uses the sockets network API and requires an Ethernet connection

that supports TCP/IP. The Tornado environment must be configured to provide these capabilities. See “Installing the Wind River Systems Tornado Software” on page 34-10 for more information.

To configure your application to use external mode, select **External mode** in the **Real-Time Workshop > Tornado Target** pane. You cannot select both **External mode** and **StethoScope** at the same time. Two **Host/Target interface** options appear below the **External mode** check box: **Transport layer** and **MEX-file arguments**.

The value of **Transport layer** is preset to `tcpip` and cannot be changed; the Tornado environment supports only the TCP/IP protocol. To the right of the **Transport layer** field, the string `MEX-file name: ext_comm` names the MEX file that the Simulink external mode transport layer uses with the Tornado target. This name cannot be changed. The full path of the MEX-file is:

```
matlabroot/toolbox/rtw/rtw/ext_comm.*
```

where `*` is a host-dependent MEX-file extension.

The value of **MEX-file arguments** must be a string of arguments to be passed to the transport layer MEX-file. The format is:

```
'target_network_name' [verbosity] [TCP port number]
```

You must provide the first argument, which specifies the name of the VxWorks target system. You can also specify a verbosity and a TCP port number. Verbosity can be 0 (the default) or 1 if extra information is desired. The TCP port number can range from 256 to 65535; the default is 17725. If a conflict exists with other software using port 17725, you can change the port with the third argument. The arguments are positional, so you must specify a verbosity if you specify a port number.

For example, to specify a target system called `halebopp`, the default verbosity of 0, and the nondefault TCP port 18000, give **MEX-file arguments** the value:

```
'halebopp' 0 18000
```

You can also enable verbose mode by using the OPTS flag to add -DVERBOSE to the **Make command** shown on the **Real-Time Workshop > General** pane. For example:

```
make_rtw OPTS="-DVERBOSE"
```

Configuring the Template Makefile

In addition to setting code generation options, you must enter some information into the VxWorks makefile, which by default is named `tornado.tmf`. You can rename the makefile if you like, provided that you also specify the name as a code generation parameter, as described in “Selecting the Tornado Target” on page 34-13.

The makefile information you must enter describes the environment in which you are using the VxWorks system. This section lists the lines in the file that you must edit.

Specifying the VxWorks Target Type and CPU

To provide information used by the VxWorks system, you must specify the type of target and the specific CPU on the target. The target type is then used to locate the correct cross-compiler and linker for your system.

The CPU type is used to define the CPU macro that is in turn used by many of the VxWorks header files. See the *VxWorks Programmer’s Guide* for information on the appropriate values to use.

This information is in the section labeled

```
#----- VxWorks Configuration -----
```

Edit the following lines to reflect your setup.

```
VX_TARGET_TYPE = 68k
CPU_TYPE = MC68040
```

Specifying the Tornado Tool Locations

To locate the Tornado tools used in the build process, the following three macros must either be defined in the environment or specified in the template makefile. Modify these macros to reflect your setup.

```
#----- Tool Locations -----
WIND_BASE = c:/Tornado
WIND_REGISTRY = $(COMPUTERNAME)
WIND_HOST_TYPE = x86-win32
```

Note On a Windows system, if the path you specify to the WIND_BASE macro contains a directory name starting with lowercase or uppercase t, n, or r, use Windows backslash notation to specify the path. For example, specify C:\Tornado_vxWorks instead of C:/Tornado_vxWorks.

During an automatic download, the software converts an instance of /t, /n, or /r in the WIND_BASE path to a form that is interpreted by a Perl script as representing a whitespace character, such as \t (tab), \n (newline), or \r (return). The software flags an error, usually stating that the volume letter (such as the C: in C:/Tornado_vxWorks) is not recognized or cannot be found. Using Windows backslash notation in the WIND_BASE path avoids this error.

Resolving Header File Paths

Tornado version 2.2.1 installs some standard header files in an include directory under the target compiler target directory. For example, if you are targeting the Freescale 68xxx processor for the VxWorks system with the GCC 2.96 compiler, the Tornado environment installs the header files at the following location:

```
WIND_BASE/host/WIND_HOST_TYPE/lib/gcc-lib/m68k-wrs-vxworks/gcc-2.96/include
```

To use Tornado version 2.2.1 or higher with the Tornado (VxWorks) Real-Time Target, `tornado.tlc`, you must enable a macro in template makefile `tornado.tmf`. To enable the macro,

- 1** Open `matlabroot/rtw/c/tornado/tornado.tmf`.
- 2** Search for `TORNADO_TARGET_COMPILER_INCLUDES`.
- 3** Uncomment the macro `TORNADO_TARGET_COMPILER_INCLUDES` and set it to the include directory that contains the Tornado standard header files.

Given the path shown above, you would set the macro as follows:

```
TORNADO_TARGET_COMPILER_INCLUDES =
$(WIND_BASE)/host/$(WIND_HOST_TYPE)/lib/gcc-lib/m68k-wrs-vxworks/gcc-2.96/include
```

If you are using a Tornado version before 2.2.1, leave the macro commented out.

Configuring for Automatic Downloading

To perform automatic downloading during the build process, you must specify the target name and host name that the Tornado target server are to run on. Modify these macros to reflect your setup.

```
#----- Macros for Downloading to Target-----
TARGET = targetname
TGTSVR_HOST = hostname
```

Building the Application

Once you have completed the operations previously described in “Installing the Wind River Systems Tornado Software” on page 34-10 and “Implementing a Wind River Systems Tornado Application” on page 34-12, you are ready to build the real-time application:

- 1** If you specified automatic downloading, reset the Wind River Systems VxWorks system by pressing control-x on the host console or power-cycling the VxWorks chassis. This ensures that no dangling processes or stale data exist in the system.

Note Be sure to reset the VxWorks system again before every subsequent build and automatic download.

- 2** Click the **Build** button in the **Real-Time Workshop** pane of the Model Explorer or the Configuration Parameters dialog box.

The resulting object file is named with the .lo extension, which stands for *loadable object*. This file is compiled for the target processor using the cross-compiler specified in the makefile.

Automatic Download and Execution

If you specified automatic downloading, the following actions occur automatically as soon as the build is complete:

- 1** The target server is started.
- 2** The object file is downloaded.
- 3** If **StethoScope** was selected, the StethoScope object files (`libxdr.so`, `libutilstssip.so`, and `libscoope.so`) are downloaded.
- 4** The downloaded program is started on the target.

If **StethoScope** was selected, you can now start StethoScope on the host. See the *StethoScope User's Manual* for more information.

Manual Download and Execution

In this section...

- “Introduction” on page 34-24
- “Resetting Wind River Systems VxWorks System” on page 34-24
- “Downloading Files” on page 34-24
- “Starting the Program” on page 34-25

Introduction

If you did not specify automatic downloading, you must use Wind River Systems Tornado tools to download and start the program.

Resetting Wind River Systems VxWorks System

Reset the VxWorks system by pressing control-x on the host console or power-cycling the VxWorks chassis. This ensures that no dangling processes or stale data exist in the system.

Note Be sure to reset the VxWorks system again before every subsequent manual download.

Downloading Files

To download the real-time program, use the VxWorks `ld` routine from within the Wind River Systems WindSh™ (wind shell). The WindSh shell can also be run from the command line or from within the Tornado development environment.

For example, if you want to download the file `vx_equal.lo`, which is in the `/home/my_working_dir` directory, use the following commands at the WindSh prompt.

```
cd "/home/my_working_dir"  
ld <vx_equal.lo
```

You will also need to load the StethoScope libraries if the **StethoScope** option was selected during the build. The *Tornado User's Guide* describes the `ld` library routine.

Starting the Program

The real-time program defines a function, `rt_main()`, which spawns the tasks to execute the model code, and communicate with the Simulink engine if you selected external mode during the build procedure. It also initializes StethoScope if you selected this option during the build procedure.

The `rt_main` function is defined in the `rt_main.c` application module. This module is located in the `matlabroot/rtw/c/tornado` directory.

The `rt_main` function takes six arguments, and is defined by the following ANSI²⁷ C function prototype:

```
RT_MODEL * (*model_name)(void),
char_T      *optStr,
char_T      *scopeInstallString,
int_T       scopeFullNames,
int_T       priority,
int_T       port
```

The following table lists the arguments to this function.

Arguments to the `rt_main` RT_MODEL

Argument	Description
<code>model_name</code>	Pointer to the entry point function in the generated code. This function has the same name as the Simulink model. It registers the local functions that implement the model code by adding function pointers to the model's <code>rtM</code> .

27. ANSI® is a registered trademark of the American National Standards Institute, Inc.

Arguments to the rt_main RT_MODEL (Continued)

Argument	Description
optStr	<p>Options string used to specify a stop time (-tf) and whether to wait (-w) in external mode for a message from the Simulink engine before starting the simulation. An example options string is</p> <pre>" -tf 20 -w"</pre> <p>The -tf option overrides the stop time that was set during code generation. If the value of the -tf option is inf, the program runs indefinitely.</p>
scopeInstallString	<p>Character string that determines which signals are installed to StethoScope. Possible values are</p> <ul style="list-style-type: none"> • NULL — Install no signals. This is the default value. • "*" — Install all signals. • "[A-Z]*" — Install signals from blocks whose names start with an uppercase letter. <p>Specifying any other string installs signals from blocks whose names start with that string.</p>
scopeFullNames	<p>Determines whether StethoScope uses full hierarchical block names for the signals it accesses or just the individual block name. Possible values are</p> <ul style="list-style-type: none"> • 1 Use full block names. • 0 Use individual block names. This is the default value. <p>It is important to use full block names if your program has multiple instances of a model or S-function.</p>

Arguments to the `rt_main` RT_MODEL (Continued)

Argument	Description
<code>priority</code>	Priority of the program's highest priority task (<code>tBaseRate</code>). Not specifying any value (or specifying a value of 0) assigns <code>tBaseRate</code> to the default priority, 30.
<code>port</code>	Port number that the external mode sockets connection should use. The valid range is 256 to 65535. The port number defaults to 17725.

Passing optStr by Using the Template Makefile

You can also pass the `-w` and `-tf` options (see `optStr` in the preceding table) to `rt_main` by using the `PROGRAM_OPTS` macro in `tornado.tmf`. `PROGRAM_OPTS` passes a string of the form

```
-opt1 val1 -opt2 val2
```

In the following examples, the `PROGRAM_OPTS` directive sets an infinite stop time and instructs the program to wait for a message from the Simulink engine before starting the simulation. The argument string must be delimited by single quotes nested within double quotes:

```
PROGRAM_OPTS = "'-tf inf -w'"
```

Including the single quotes ensures that the argument string is passed to the target program correctly under both Microsoft Windows and The Open Group UNIX platforms.

Calling `rt_main`

To begin program execution, call `rt_main` from the WindSh prompt. For example,

```
sp(rt_main, vx_equal, "-tf 20 -w", "*", 0, 30, 17725)
```

- Begins execution of the `vx_equal` model
- Specifies a stop time of 20 seconds

- Provides access to all signals (block outputs) in the model by StethoScope
- Uses only individual block names for signal access (instead of the hierarchical name)
- Uses the default priority (30) for the **tBaseRate** task
- Uses TCP port 17725, the default

Interacting with Target Application Data Using the C API

The C API allows you to write host-based or target-based code that interacts with signals, states, and parameters in your target-based application code.

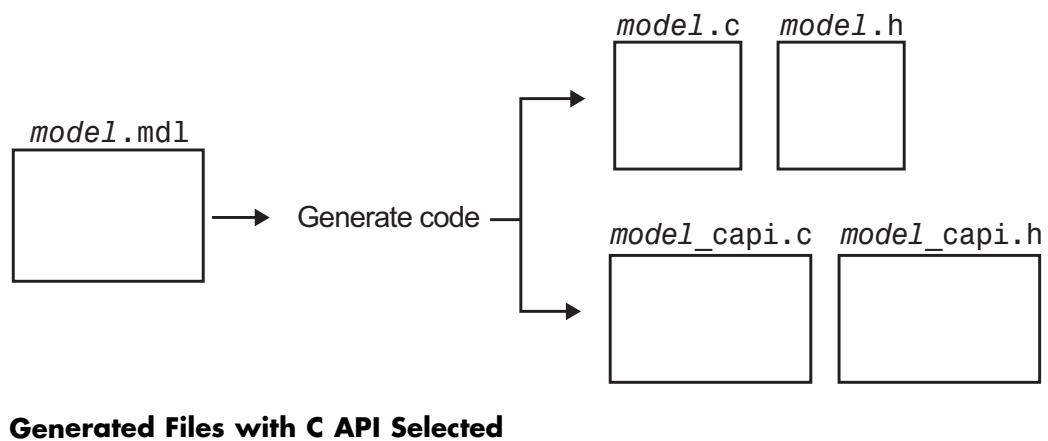
- “Introduction” on page 35-2
- “Generating C API Files” on page 35-3
- “Description of C API Files” on page 35-5
- “Using the C API in an Application” on page 35-21
- “C API Limitations” on page 35-33

Introduction

Some Real-Time Workshop applications need to interact with signals, states, or parameters in a model's generated code. For example, calibration applications monitor and modify parameters. Signal monitoring or data logging applications interface with signal and state data. Using the Real-Time Workshop C API, you can build target applications that log signals and states, monitor signals and states, and tune parameters, while the generated code executes.

The C API uses a small memory footprint, which is achieved by sharing information common to signals, states, and parameters in smaller structures. An index into the structure map is provided in the signal, state, or parameter structure, allowing multiple signals, states, or parameters to share data.

When you configure a model to use the C API, the Real-Time Workshop 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 directory, based on settings on the Configuration Parameters dialog box. The C API source code file contains information about global block output signals, states, 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.



Generating C API Files

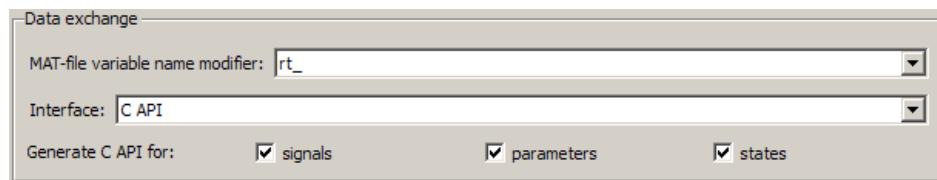
To generate C API files for your model,

- 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.
 - “Selecting C API with the Configuration Parameters Dialog Box” on page 35-3
 - “Selecting C API from the MATLAB Command Line” on page 35-4
- 2** Generate code for your model.

After generating code, you can examine the files *model_capi.c* (or *.cpp*) and *model_capi.h* in the model build directory.

Selecting C API with the Configuration Parameters Dialog Box

- 1** Open your model, and launch either the Configuration Parameters dialog box or Model Explorer.
- 2** Go to the **Real-Time Workshop > Interface** pane and, in the **Data exchange** subpane, select C API as the value for the **Interface** parameter. The **Generate C API for: signals**, **Generate C API for: parameters**, and **Generate C API for: states** check boxes appear, as shown in the next figure.



- 3** 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 select all three check boxes, signals, parameters, and states will appear in the C API generated code.

Selecting C API from the MATLAB 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. Enter one or more of the following MATLAB commands, where `modelname` is the name of your model:

To select **Generate C API for: signals**, type

```
set_param('modelname','RTWCAPISignals','on')
```

To clear **Generate C API for: signals**, type

```
set_param('modelname','RTWCAPISignals','off')
```

To select **Generate C API for: parameters**, type

```
set_param('modelname','RTWCAPIParams','on')
```

To clear **Generate C API for: parameters**, type

```
set_param('modelname','RTWCAPIParams','off')
```

To select **Generate C API for: states**, type

```
set_param('modelname','RTWCAPIS States','on')
```

To clear **Generate C API for: states**, type

```
set_param('modelname','RTWCAPIS States','off')
```

Generating C API and ASAP2 Files

The C API and ASAP2 interfaces are not mutually exclusive. Although the **Interface** option on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box allows you to select either the ASAP2 or C API interface, you can instruct the Real-Time Workshop code generator to generate files for both interfaces. For details, see “Generating ASAP2 and C API Files” on page 36-21.

Description of C API Files

In this section...

- “Overview” on page 35-5
- “Structure Arrays Generated in C API Files” on page 35-8
- “Generating Example C API Files” on page 35-9
- “C API Signals” on page 35-11
- “C API States” on page 35-15
- “C API Parameters” on page 35-16
- “Mapping C API Data Structures to the Real-Time Model Data Structure” on page 35-19

Overview

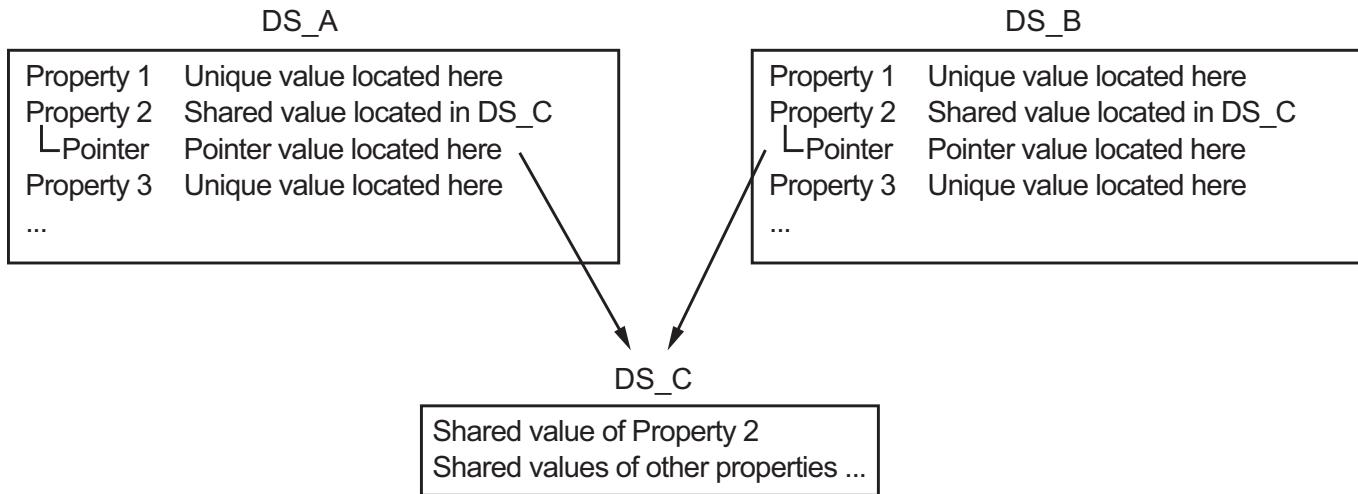
The *model_capi.c* (or *.cpp*) file provides external applications with a consistent interface to the model’s data. Depending on your configuration settings, the data could be a signal, state, or parameter. In this discussion, the term “data item” refers to either a signal, a state, or a parameter. The 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 there are multiple data items in the model, the interface generates an array of data structures. The members of a data structure map to data properties.

Typically, to interface with data items, an application requires the following properties for each:

- Name
- Block path
- Port number (for signals 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 and states 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 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 data item's structure. So 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 fact that some data items can 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 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.

Array of Signal Structures

```
rtwCapi_Signals rtBlockSignals[]
{
{
...
blockPath      sys/blk1
signalName    signal1
portNumber     0
dataTypeIndex  0
...
},
{
...
blockPath      sys/blk2
signalName    signal2
portNumber     1
dataTypeIndex  0
...
},
{
...
blockPath      sys/blk3
signalName    signal3
portNumber     0
dataTypeIndex  1
...
}
};
```

Array of Data Type Structures

```
rtwCAPI_DataTypeMap rtDataTypeMap[]
{
{
cName          "double"
mwName         "real_T"
numElements    0
elemMapIndex   0
dataSize       sizeof(real_T)
s1DataId       SS_DOUBLE
isComplex      0
isPointer      0
};
{
cName          "int"
mwName         "int32_T"
numElements    0
elemMapIndex   0
dataSize       sizeof(int32_T)
s1DataId       SS_INT32
...
}
};
```

These indices of 0 point to the first element in the rtDataTypeMap array.

The index of 1 points to the second element in the rtDataTypeMap array.

The figure shows three signals. Notice that **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, and parameters, so states 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 data item's structure. 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 directory). 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's 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's 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's name, block path, type (continuous or discrete), and indices to the address, data type, dimension, fixed-point, and sample-time structure arrays.
- **rtModelParameters** is an array of structures that contains information about all 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

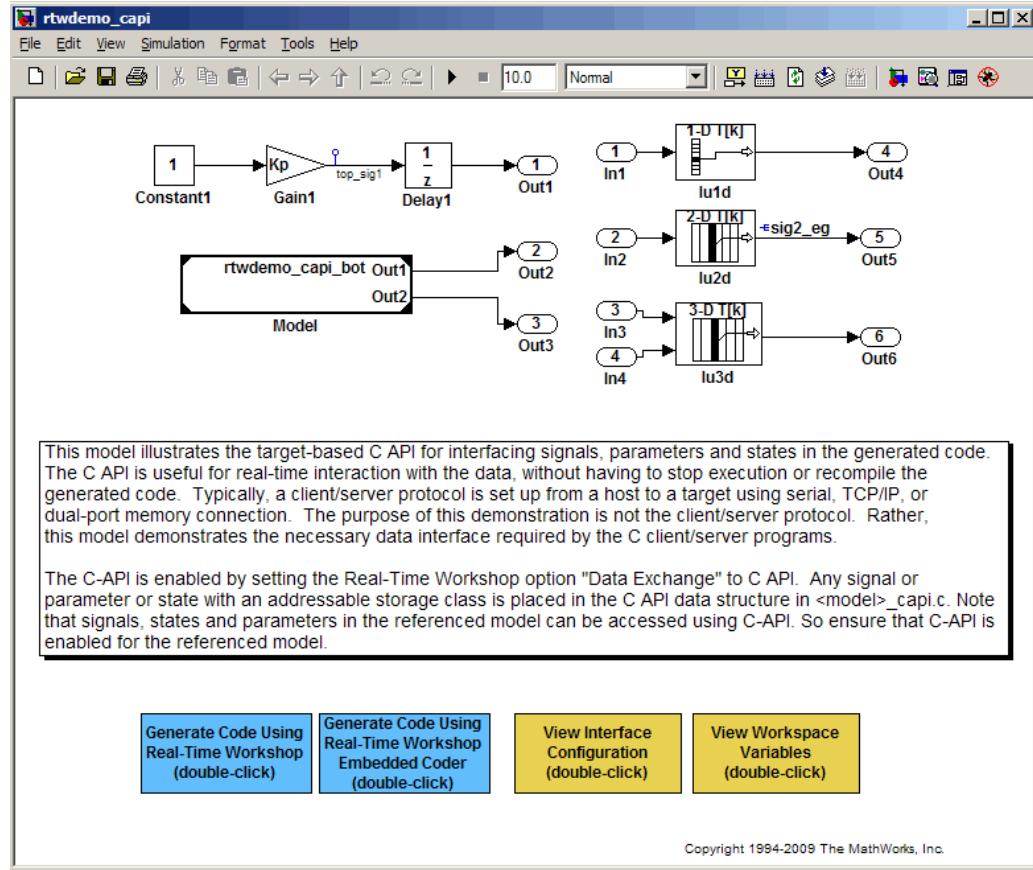
provide the variable's name, address, and indices to data type, dimension, and fixed-point structure arrays.

- **rtDataAddrMap** is an array of base addresses of signals, states, 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, and parameters. Each element of this array is of type `struct rtwCAPI_FixPtMap`. The members of this structure provide information about the data's scaling, bias, exponent, and whether or not the fixed-point data is signed. If the model does not have fixed-point data (signal, state, or parameter), the Real-Time Workshop 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 model's global signals and states. (This array contains no 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.

Generating Example C API Files

The next three sections, “C API Signals” on page 35-11, “C API States” on page 35-15, and “C API Parameters” on page 35-16, discuss generated C API structures using the demo model `rtwdemo_capi` as an example. To generate code from the demo model, do the following:

- 1 Open the model by clicking the link above or by typing `rtwdemo_capi` on the MATLAB command line. The model appears as shown in the next figure.



- 2 Generate code for the model by double-clicking the button **Generate Code Using Real-Time Workshop**.

Note The C API code examples in the next three sections were generated with C selected 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 submodel `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 submodel

Additionally, the model has five global block parameters that will appear in C API generated code:

- `Kp` (top model Gain1 block and submodel Gain2 block share)
- `Ki` (submodel 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's 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, ffpIndex, 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, be sure to read the file's comments. For example, notice the comment that begins on the third line above. 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 this 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`. (It 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. So, from the `rtDataAddrMap` array, you can infer that the address of this signal is `&sig2_eg[0]`.

This level of indirection is provided to support 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 “Setting Up Support for Code Reuse” and “Model Entry Points” in the Real-Time Workshop Embedded Coder documentation.)

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`, it points to the first structure element in the array. So you can infer that the signal’s data type 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. This saves 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`, it points to the second element in the `rtDimensionMap` array:

```
/* Dimension Map - use dimensionMapIndex to access elements of this 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 any fixed-point information about the signal. Your code can use the scaling information provided 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 Toolbox documentation.)

Because this index is 0 for `sig2_eg`, the signal has no fixed-point information. A fixed-point map index of zero always means that the signal has no 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. The sampling information can be useful if you log multirate signals or conditionally executed signals.

Note `model_capi.c` (or `.cpp`) includes `rtw_capi.h`. Any source file that references the `rtBlockSignals` array also should include `rtw_capi.h`.

C API States

The `rtwCAPI_States` structure captures state information including the state's 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 35-11, as follows:

- The address of the signal is given by `addrMapIndex`, which, in this example, is 2. This is an index into the `rtDataAddrMap` array, found later in `rtwdemo_capi_capi.c`. Because the index is zero based, 2 corresponds to the third element in `rtDataAddrMap`, which is `&rtwdemo_capi_DWork.top_state`.
- The `dataTypeIndex` provides an index into the `rtDataTypeMap` array, found later in `rtwdemo_capi_capi.c`, indicating the data type of the parameter. The value 0 corresponds to a double, noncomplex parameter.
- The `dimIndex` (dimensions index) provides an index into the `rtDimensionMap` array, found later in `rtwdemo_capi_capi.c`. The value 0 corresponds to the first entry, which is `{ rtwCAPI_SCALAR, 0, 2, 0 }`.
- The `fixPtIndex` (fixed-point index) provides an index into the `rtFixPtMap` array, found later in `rtwdemo_capi_capi.c`, indicating any fixed-point information about the parameter. As with the corresponding signal attribute, a fixed-point map index of zero always means that the parameter has no fixed-point information.

C API Parameters

The `rtwCAPI_BlockParameters` and `rtwCAPI_ModelParameters` structures capture parameter information including the parameter’s 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** 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 **Inline parameters** is cleared,

- 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 Real-Time Workshop software assigns its elements only `NULL` or zero values in the absence of such data.
- If **Inline parameters** is selected,
 - The `rtBlockParameters` array is empty. The Real-Time Workshop software assigns its elements only `NULL` or zero values.
 - The `rtModelParameters` array contains entries for all 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` demo 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 35-11, 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 any fixed-point information about the parameter. As with the corresponding signal attribute, a fixed-point map index of zero always means that the parameter has no fixed-point information.

Mapping C API Data Structures to the Real-Time Model Data Structure

The real-time model data structure encapsulates model data and associated information necessary to describe the model fully. When you select the C API feature and generate code, the Real-Time Workshop 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;
```

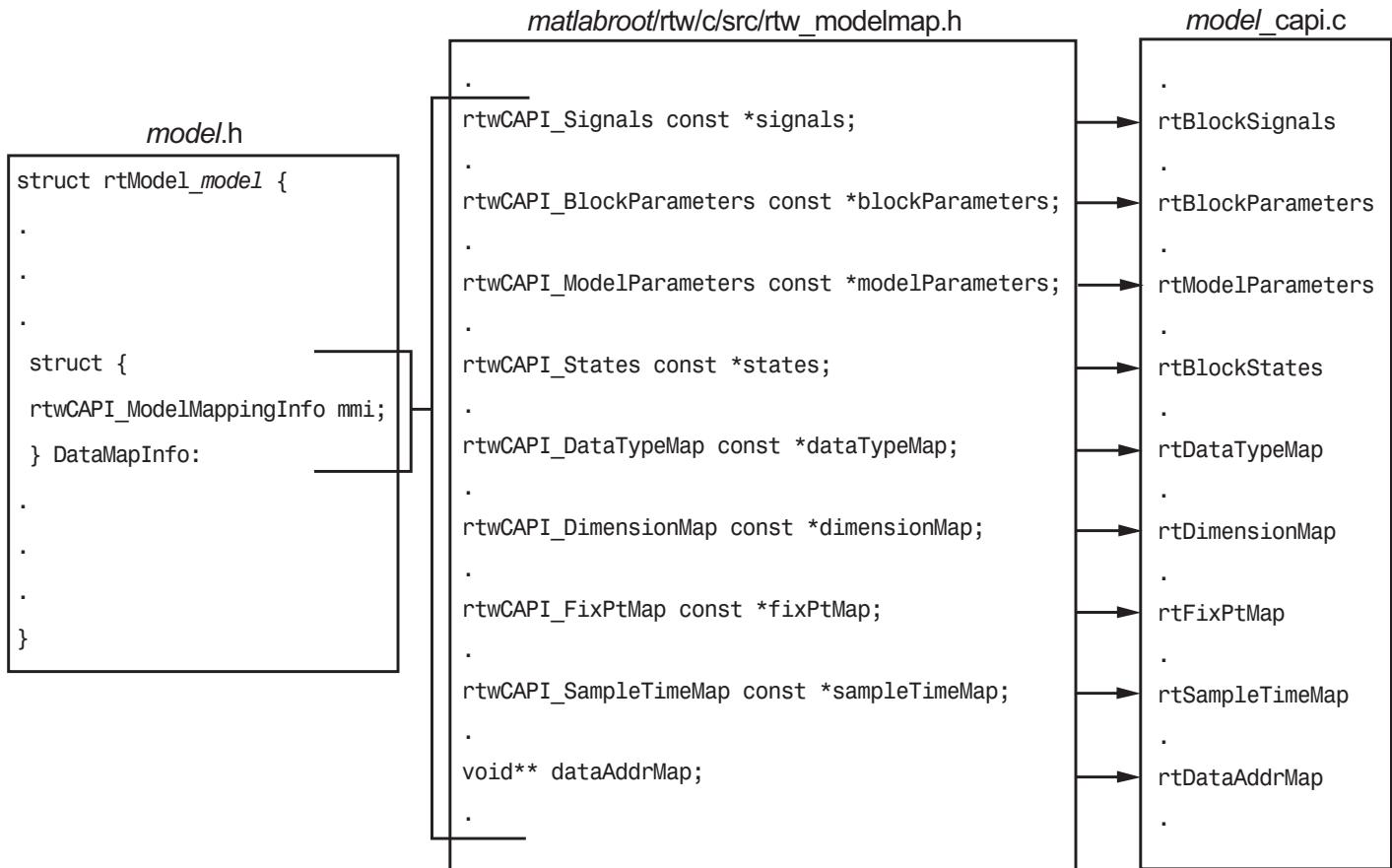
This member defines *mmi* (for model mapping information) of type *struct rtwCAPI_ModelMappingInfo*. The structure is provided in *matlabroot/rtw/c/src/rtw_modelmap.h* (where *matlabroot* represents the root of your MATLAB installation directory). The *mmi* substructure defines the interface between 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. See the next figure. 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 */
} 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 demo model *rtwdemo_capi*:

```
/* Initialize DataMapInfo substructure containing ModelMap for C API */
rtwdemo_capi_InitializeDataMapInfo(rtwdemo_capi_M);
```



Mapping Between Model and C API Arrays of Structures

Note This figure lists the arrays in the order their structures appear in *rtw_modelmap.h*, which differs slightly from their generated order in *model_capi.c*.

Using 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, and parameters. Your target-based application code is compiled with the Real-Time Workshop 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 directory), 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, and parameters.

In this section...

- “Example: Using the C API to Access Model Signals and States” on page 35-21
- “Example: Using the C API to Access Model Parameters” on page 35-28

Example: Using the C API to Access Model Signals and States

An example application is provided below that logs a model’s global signals and states 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 directory.
 - `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 any or all of the following methods:

- **Real-Time Workshop > Custom Code** pane of the Configuration Parameters dialog box
- Custom Code library blocks
- TLC custom code functions

This tutorial uses the **Real-Time Workshop > Custom Code** pane and the Systems 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 `MdlStart` function (or if an `ert.tlc` target is selected for the model, in the `model_initialize` function).
- `capi_UpdateLogging` is called in the `model_output` 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 MdlStart(void)
{
```

```

/* user code (Start function Trailer) */

/* 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 output function */
static void rtwdemo_capi_output(int_T tid)
{
    ...

    /* user code (Output function Trailer) */
    /* 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 Trailer) */
    /* 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");
    }
}

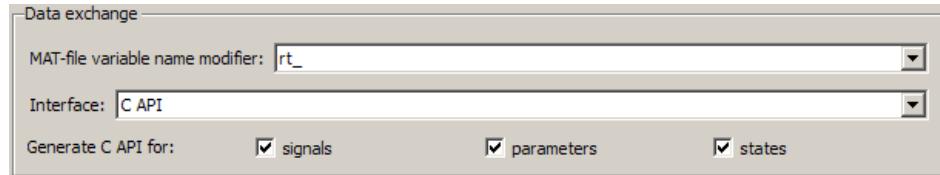
```

The following procedure illustrates how you can use the C API macro and function interfaces to log a model's global signals and states to a text file.

- 1 Open the demo model `rtwdemo_capi` by entering `rtwdemo_capi` at the MATLAB command line.
- 2 Open the Configuration Parameters dialog box and go to the **Real-Time Workshop** pane.
- 3 For the **System target file** parameter, select `grt.tlc`. (Alternatively, if you are licensed for Real-Time Workshop Embedded Coder software, you can select `ert.tlc`. Make sure that you also select `ert.tlc` for the referenced model `rtwdemo_capi_bot`.)

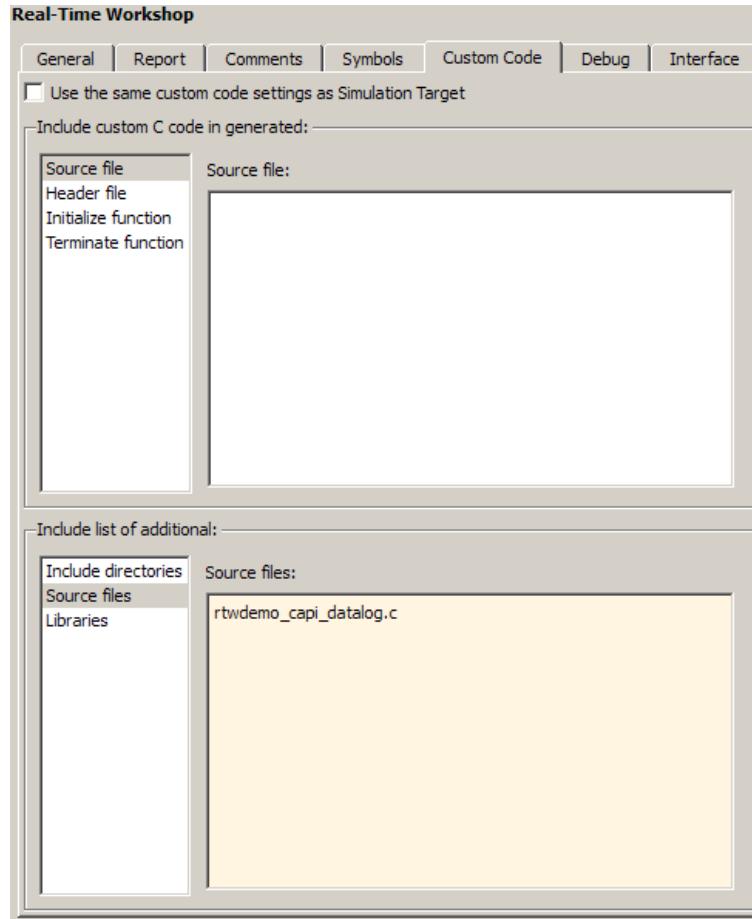
Note Selecting a system target file other than `grt.tlc` or disabling some C API options (signals, parameters, or states) in the top model requires corresponding changes in the referenced model. Since the demo models have read-only access, save the updated referenced model with a different name and modify the top model to reference the renamed model.

- 4 Go to the **Interface** pane.
 - a In the **Data exchange** subpane, for the **Interface** parameter, verify that **C API** is selected.
 - b Additionally, verify that the options **Generate C API for: signals** and **Generate C API for: states** are selected. This example also leaves **Generate C API for: parameters** selected. If you clear any of the options, make sure that the settings match between the top model and the referenced model.



- c If you are using the `ert.tlc` target, verify that the options **MAT-file logging** and **Support: complex numbers** are selected.

- If you modified any option settings in this step, click the **Apply** button.
- 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 appears on the right.
- 6 In the **Include directories** field, type `matlabroot/toolbox/rtw/rtwdemos`, where `matlabroot` represents the root of your MATLAB installation directory.
- 7 In the **Include list of additional** subpane, click **Source files**, and type `rtwdemo_capi_datalog.c`, as shown in the figure below.



- 8** In the **Include custom C code in generated** subpane, click **Source file**, and type or copy in the following include statement:

```
#include "rtwdemo_capi_datalog.h"
```

- 9** In the **Initialize function** field, type or copy in 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);  
}
```

- 10** In the **Terminate function** field, type or copy in 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");  
}
```

- 11** Click the **Apply** button.

- 12** Open the Real-Time Workshop Custom Code library by entering `custcode` in the MATLAB command window. Add a System Outputs block to the top level of the `rtwdemo_capi` model.

- 13** Double-click the System Outputs block to open the System Outputs Function Custom Code dialog box. In the field **System Outputs Function Exit Code**, type or copy in 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));
}

```

Click the **OK** button.

- 14** In the **Real-Time Workshop** pane, verify that the **Build** button is visible. If necessary, clear the option **Generate code only** and click the **Apply** button.

Click the **Build** button to build the model and generate an executable file. For example, on a Windows system, the build generates the file `rtwdemo_capi.exe`.

- 15** To run the executable file, enter the command `!rtwdemo_capi` in the MATLAB command window. During execution, signals and states are logged using the C API and then written to the text file `rtwdemo_capi_ModelLog.txt` in your working directory.

```

>> !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 **

```

- 16** Examine the text file in the MATLAB editor or any text editor. An excerpt of the signal and state logging output is shown below.

```

***** 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
...
...
```

Example: Using the C API to Access Model Parameters

An example application is provided below that prints the parameter values of all 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 all 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;

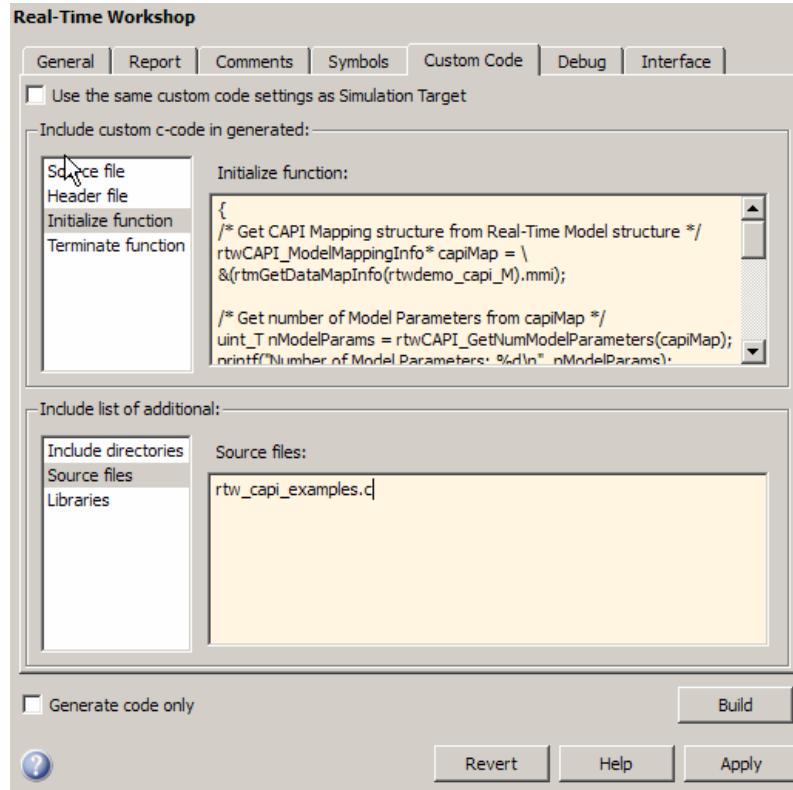
    for (idx=0; idx < nModelParams; idx++) {
        /* call print utility function */
        capi_PrintModelParameter(capiMap, idx);
    }
}
```

```
}
```

The print utility function is provided in `matlabroot/rtw/c/src/rtw_capi_examples.c` (where `matlabroot` represents the root of your MATLAB installation directory). This file contains utility functions for accessing the C API structures.

To become familiar with the example code, try building a model that displays all the tunable block parameters and MATLAB variables. You can use `rtwdemo_capi.mdl`, the C API demo model, for this purpose. The steps below apply to both `grt.tlc` and `ert.tlc` targets, unless otherwise indicated:

- 1** Open the model `rtwdemo_capi.mdl` by typing `rtwdemo_capi` at the MATLAB command line.
- 2** Open the Configuration Parameters dialog box and go to the **Optimization** pane.
- 3** Verify that the **Inline parameters** option is selected.
- 4** If you want to use the `ert.tlc` target instead of the default `grt.tlc`, select an `ert.tlc` target in the **System target file** field on the **Real-Time Workshop** pane and click **OK**.
- 5** Use the **Custom Code** pane to embed your custom application code in the generated code. Select the **Custom Code** pane, and then click **Initialize function**. The **Initialize function** input field appears on the right.
- 6** In the **Initialize function** input field, type or copy the example application code provided above. This embeds the application code in the `MdlStart` function. (If you are using `ert.tlc`, the code appears in the `model_initialize` function.)
- 7** Click **Include directories**, and type `matlabroot/rtw/c/src`, where `matlabroot` represents the root of your MATLAB installation directory.
- 8** In the **Include list of additional** subpane, click **Source files**, and type `rtw_capi_examples.c`, as shown in the figure below. Click the **Apply** button.



- 9 If you are using the `ert.tlc` target, select the following options on the **Real-Time Workshop > Interface** pane, and then click **Apply**:
 - **C API** in the **Interface** list
 - **MAT-file logging**
 - **Support: complex numbers**
- 10 Go to the **Real-Time Workshop** pane and clear the **Generate code only** check box if it is not already cleared.
- 11 Click the **Build** button. The Real-Time Workshop code generator generates the executable file `rtwdemo_capi.exe` in your current working directory.
- 12 Type `!rtwdemo_capi` at the MATLAB command line to run the executable file. Parameter information is displayed in the Command Window, as shown below.

```
>> !rtwdemo_capi

** starting the model **
Number of Model Parameters: 5
Ki =
    7
Kp =
    4
p1 =
    0.8147
    0.9058
    0.127
p2 =
    0.9649 0.9706 0.4854
    0.1576 0.9572 0.8003
p3 =
ans(:,:,1) =
    0.1419 0.9157 0.9595 0.03571
    0.4218 0.7922 0.6557 0.8491

ans(:,:,2) =
    0.934 0.7577 0.3922 0.1712
    0.6787 0.7431 0.6555 0.706

>>
```

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.
- The following signals are not supported:
 - External inputs
 - External outputs
 - Local block outputs
- Local Stateflow parameters are not supported.
- The following custom storage class objects are not supported:
 - Objects without the package `csc_registration` file are not supported.
 - `BitPackBoolean` objects, grouped custom storage classes, and objects defined by using macro are not supported.
- 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 take effect in code generation only if you use the ERT target and clear the **Ignore custom storage classes** check box on the Configuration Parameters dialog box.

Generating Model Information for Host-Based 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 standard description you use for data measurement, calibration, and diagnostic systems.

- “Overview” on page 36-2
- “Targets Supporting ASAP2” on page 36-3
- “Defining ASAP2 Information” on page 36-4
- “Generating an ASAP2 File” on page 36-7
- “Customizing an ASAP2 File” on page 36-12
- “Structure of the ASAP2 File” on page 36-19
- “Generating ASAP2 and C API Files” on page 36-21

Overview

The Real-Time Workshop 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.de>.
- Simulink data objects. Data objects are used to supply information not contained in the model. For an overview, see “Working with Data” in the Simulink documentation.
- Storage and representation of signals and parameters in generated code. See Defining Data Representation and Storage for Code Generation on page 1.
- Signal and parameter objects and their use in code generation. See Defining Data Representation and Storage for Code Generation on page 1.

If you are reading this document online in the MATLAB Help browser, you can run an interactive demo of ASAP2 file generation.

Alternatively, you can access the demo by typing the following command at the MATLAB command prompt:

```
rtdemo_asap2
```

Targets Supporting ASAP2

ASAP2 file generation is available to all Real-Time Workshop target configurations. You can select these target configurations from the System Target File Browser. For example,

- The **Generic Real-Time Target** lets you generate an ASAP2 file as part of the code generation and build process.
- Any of the **Real-Time Workshop Embedded Coder** target selections also lets you generate an ASAP2 file as part of the code generation and build process.
- The **ASAM-ASAP2 Data Definition Target** lets you generate only an ASAP2 file, without building an executable.

Procedures for generating ASAP2 files by using these target configurations are given in “Generating an ASAP2 File” on page 36-7.

Defining ASAP2 Information

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 with the necessary properties.

You can use built-in Simulink data objects to provide the necessary information. For example, you can use `Simulink.Signal` objects to provide information about **MEASUREMENTS** and `Simulink.Parameter` objects to provide information about **CHARACTERISTICS**. Also, you can use data objects from data classes that are derived from `Simulink.Signal` and `Simulink.Parameter` to provide the necessary information. For details, see "Working with Data" in the Simulink documentation.

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	<code>Description</code>
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	<code>Min</code>
Maximum allowable value	Data object	<code>Max</code>

Data Attribute	Defined In	Property Name
Units	Data object	DocUnits
Memory address (optional)	Data object	MemoryAddress_ASAP2 (optional; see “Memory Address Attribute” on page 36-5.)

Memory Address Attribute

If the memory address attribute is unknown before code generation, a placeholder string is inserted 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.

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 `ASAP2.Signal` or `ASAP2.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 “Subclassing Simulink Data Classes” in the Simulink documentation.

Note In previous releases, for ASAP2 file generation, it was necessary 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

Generating an ASAP2 File

In this section...

“Introduction” on page 36-7

“Using Generic Real-Time Target or Embedded Coder Target” on page 36-7

“Using the ASAM-ASAP2 Data Definition Target” on page 36-9

“Generating ASAP2 Files for Referenced Models” on page 36-11

Introduction

You can generate an ASAP2 file from your model in one of the following ways:

- Use the Generic Real-Time Target or a Real-Time Workshop 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 demo, `rtdemo_asap2`.

Using Generic Real-Time Target or Embedded Coder Target

The procedure for generating a model’s data definition in ASAP2 format using the Generic Real-Time Target or a Real-Time Workshop Embedded Coder target is as follows:

- 1 Create the desired model. Use appropriate parameter names and signal labels to refer to CHARACTERISTICS and MEASUREMENTS, 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.

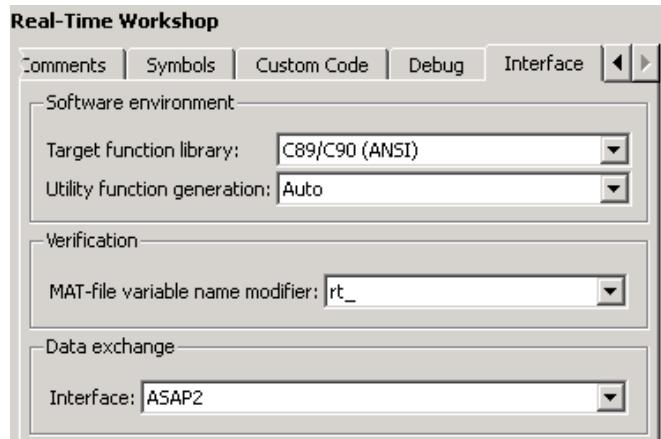
- 3 For each data object, configure the **Storage class** property to a setting other than **Auto** or **SimulinkGlobal**. This ensures that the data object is declared as global in the generated code. For example, a storage class setting of **ExportedGlobal** configures the data object as unstructured 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 Real-Time Workshop 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** 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 **Real-Time Workshop** pane, click **Browse** to open the System Target File Browser. In the browser, select **Generic Real-Time Target** or any embedded real-time target and click **OK**.
- 7 In the **Interface** field on the **Interface** pane, select **ASAP2**. The next figure shows the **Interface** pane when the **Generic Real-Time Target** is selected. If you select an embedded real-time target, the **Interface** pane looks different.



8 Select the **Generate code only** check box on the **Real-Time Workshop** pane.

9 Click **Apply**.

10 Click **Generate code**.

The Real-Time Workshop code generator writes the ASAP2 file to the build directory. By default, the file is named *model.a21*, where *model* is the name of the model. The ASAP2 filename is controlled by the ASAP2 setup file. For details see “Customizing an ASAP2 File” on page 36-12.

Using 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 appropriate parameter names and signal labels to refer to **CHARACTERISTICS** and **MEASUREMENTS**, 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 ensures that the data object is 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 Real-Time Workshop 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** 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 **Real-Time Workshop** 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 **Real-Time Workshop** pane.
- 8 Click **Apply**.
- 9 Click **Generate code**.

The Real-Time Workshop code generator writes the ASAP2 file to the build directory. By default, the file is named *model.a21*, where *model* is the name of the model. The ASAP2 filename is controlled by the ASAP2 setup file. For details see “Customizing an ASAP2 File” on page 36-12.

Generating 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 objects 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 a Real-Time Workshop Embedded Coder target, follow the procedure described in “Using Generic Real-Time Target or Embedded Coder Target” on page 36-7 for the top model and each referenced model.

Customizing an ASAP2 File

In this section...

“Introduction” on page 36-12

“ASAP2 File Structure on the MATLAB Path” on page 36-12

“Customizing the Contents of the ASAP2 File” on page 36-13

“ASAP2 Templates” on page 36-14

Introduction

The Real-Time Workshop Embedded Coder product provides a number of 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 directories identified below:

- TLC files for generating ASAP2 file

The *matlabroot/rtw/c/tlc/mw* directory contains TLC files that generate ASAP2 files, *asamlib.tlc*, *asap2lib.tlc*, and *asap2main.tlc*. These files are included by the selected **System target file**. (See “Targets Supporting ASAP2” on page 36-3.)

- ASAP2 target files

The *matlabroot/toolbox/rtw/targets/asap2/asap2* directory contains the ASAP2 system target file and other control files.

- Customizable TLC files

The *matlabroot/toolbox/rtw/targets/asap2/asap2/user* directory 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* directory contains templates that define each type of **CHARACTERISTIC** in the ASAP2 file.

Customizing 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` directory. This section refers to this directory as the `asap2/user` directory.

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` directory before making any modifications.
- 2** Remove the old `asap2/user` directory from the MATLAB path, or add the new `asap2/user` directory to the MATLAB path above the old directory. This ensures that the MATLAB session uses the ASAP2 setup file, `asap2setup.tlc` (new for Release 14).

`asap2setup.tlc` specifies the directories and files to include in the TLC path during the ASAP2 file generation process. Modify `asap2setup.tlc` to control the directories 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_LAYOUTS`: modify appropriate parts of the ASAP2 template files.

CHARACTERISTICS: modify appropriate parts of the ASAP2 template files. For more information on modifying the appearance of **CHARACTERISTICS**, see “ASAP2 Templates” on page 36-14.

- **MEASUREMENTS:** These are specified in `asap2userlib.tlc`.
- **COMPU_METHODS:** These are specified in `asap2userlib.tlc`.

ASAP2 Templates

The appearance of **CHARACTERISTICS** in the ASAP2 file is controlled using a different template for each type of **CHARACTERISTIC**. The `asap2/user` directory 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 Define a parameter group. See “Defining Parameter Groups” on page 36-14.
- 2 Create a template definition file. See “Creating Template Definition Files” on page 36-16.
- 3 Include the template definition file in the TLC path. The path is specified in the ASAP2 setup file, `asap2setup.tlc`.

Defining Parameter Groups

In some cases you must group multiple parameters together in the ASAP2 file (for example, the `x` and `y` data in a 1-D Lookup Table block). Parameter groups enable Simulink blocks to define an associative relationship among some or all of their parameters. The following example shows the `Lookup1D` parameter group and describes how to create and use parameter groups in conjunction with the ASAP2 file generation process.

The `BlockInstanceSetup` function, within a block’s TLC file, creates parameter groups. There are two built-in TLC functions that facilitate this process: `SLibCreateParameterGroup` and `SLibAddMember`. The following code creates the `Lookup1D` parameter group in `look_up.tlc`. Similar syntax is used to create parameter groups for the Lookup Table (2-D) block.

```
%if GenerateInterfaceAPI
```

```

%% Create a parameter group for ASAP2 data definition
%assign group = SLibCreateParameterGroup(block,"Lookup1D")
%assign tmpVar = SLibAddMember(block,group,InputValues)
%assign tmpVar = SLibAddMember(block,group,OutputValues)
%endif

```

ParameterGroup records are not written to the *model.rtw* file, but are included as part of the relevant block records in the compiled model. The following code shows the Lookup1D parameter group. The Lookup1D parameter group has two member records. The reference fields of these records refer to the relevant x and y data records in GlobalMemoryMap:

```

Block {
    Type      Lookup
    Name      "<Root>/Look-Up Table"
    ...
    NumParameterGroups   1
    ParameterGroup {
        Name      Lookup1D
        NumMembers 2
        Member {
            NumMembers 0
            Reference ...
        }
        Member {
            NumMembers 0
            Reference ...
        }
    }
}

```

The Lookup1D parameter group is used by the function `ASAP2UserFcnWriteCharacteristic_Lookup1D`, which is defined in the template definition file, `asap2lookup1d.tlc`. This function uses the parameter group to obtain the references to the associated x and y data records in the GlobalMemoryMap, as shown in the following code.

```

%function ASAP2UserFcnWriteCharacteristic_Lookup1D(paramGroup) \
    Output
%assign xParam = paramGroup.Member[0].Reference

```

```
%assign yParam = paramGroup.Member[1].Reference  
...  
%endfunction
```

Creating 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 appropriate 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 appropriate 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
```


Structure of the ASAP2 File

The following table outlines the basic structure of the ASAP2 file and describes the Target Language Compiler 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()>.

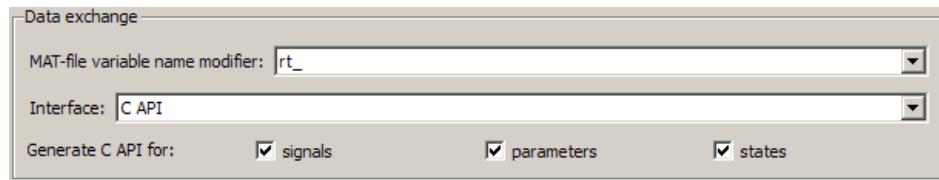
File Section	Contents of asap2main.tlc	TLC File Containing Function Definition
File header	%<ASAP2UserFcnWriteFileHead()%>	asap2userlib.tlc
/begin PROJECT ""	/begin PROJECT "%<ASAP2ProjectName>"	asap2setup.tlc
/begin HEADER "" HEADER contents	/begin HEADER "%<ASAP2HeaderName>" %<ASAP2UserFcnWriteHeader()%>	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_LAYOUTS - CHARACTERISTICS - ParameterGroups - ModelParameters	...WriteRecordLayout_TemplateName() ...WriteCharacteristic_TemplateName() ...WriteCharacteristic_Scalar()	user/templates/...
- MEASUREMENTS - ExternalInputs - BlockOutputs	...WriteMeasurement()	asap2userlib.tlc
- COMPU_METHODS	...WriteCompuMethod()	asap2userlib.tlc

File Section	Contents of asap2main.tlc	TLC File Containing Function Definition
/end MODULE	/end MODULE	
File footer/tail	%<ASAP2UserFcnWriteFileTail()>	asap2userlib.tlc

Generating ASAP2 and C API Files

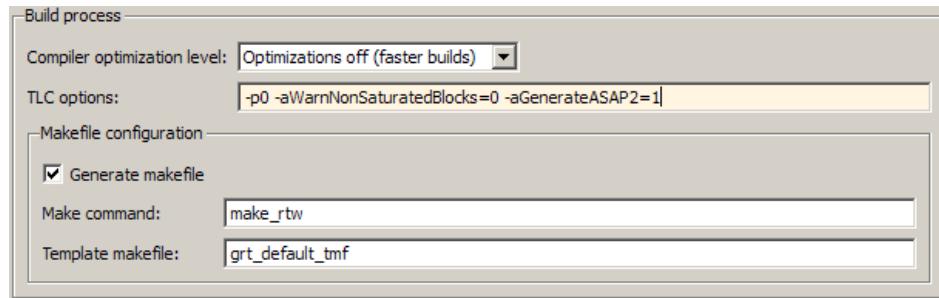
The ASAP2 and C API interfaces are not mutually exclusive. Although the **Interface** option on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box allows you to select either the ASAP2 or C API interface, you can instruct the Real-Time Workshop product to generate files for both interfaces by doing the following:

- 1 In the **Data exchange** section of the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box, select **C API** for the **Interface** option.



- 2 In the **Build process** section of the **Real-Time Workshop** pane, add the following to the **TLC options** text box:

`-aGenerateASAP2=1`



- 3 Click **Generate** or **Build**. The Real-Time Workshop code generator 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 Chapter 35, “Interacting with Target Application Data Using the C API”.

Communicating With Code Executing on a Target System Using Simulink External Mode

In external mode, the Real-Time Workshop software establishes a communications link between a model running in the Simulink environment and code executing on a target system.

- “Introduction” on page 37-3
- “Using the External Mode User Interface” on page 37-5
- “External Mode Compatible Blocks and Subsystems” on page 37-27
- “External Mode Communications Overview” on page 37-31
- “Client/Server Implementations” on page 37-35
- “Using External Mode Programmatically” on page 37-44
- “Creating a TCP/IP Transport Layer for External Communication” on page 37-50
- “External Mode Limitations” on page 37-67

“Creating a TCP/IP Transport Layer for External Communication” on page 37-50 contains information for implementing an external mode communications layer. Read it to gain increased insight into the architecture and code structure of external mode communications. Chapter 34, “Interfacing

With a Real-Time Operating System ” discusses the use of external mode in the Wind River Systems VxWorks Tornado environment.

Introduction

External mode allows two separate systems, a *host* and a *target*, to communicate. The host is the computer where the MATLAB and Simulink environments are executing. The target is the computer where the executable created by the Real-Time Workshop 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.

External mode lets you

- 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. This lets you tune your program's parameters without recompiling.
- View and log block outputs in many types of blocks and subsystems. You can monitor and/or 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 could 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 the Real-Time Workshop 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. ERT, GRT, GRT malloc, and RSim targets support external mode host/target communication by using TCP/IP and RS-232 (serial) communication. The xPC Target product uses a customized transport layer. The Wind River Systems Tornado target supports TCP/IP only. Serial transport is implemented only

for Microsoft Windows 32-bit architectures. The Real-Time Windows Target product uses shared memory.

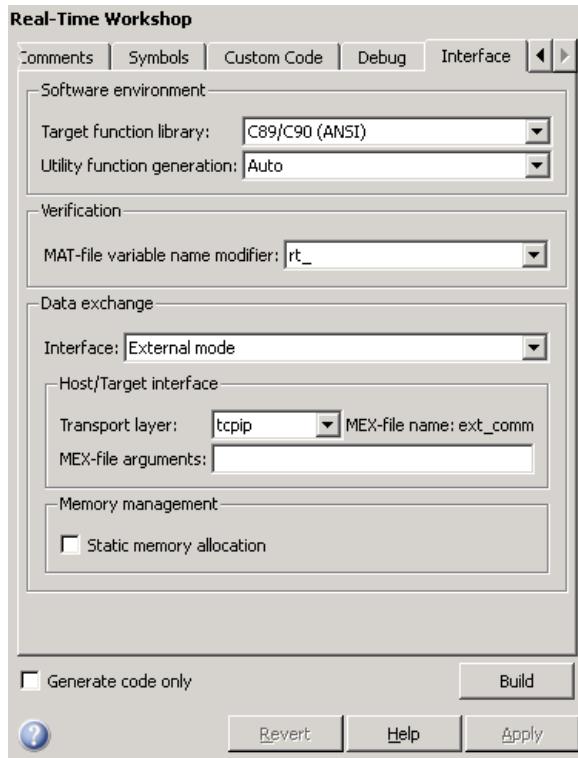
Using the External Mode User Interface

In this section...

- “External Mode Interface Options” on page 37-5
- “External Mode Related Menu and Toolbar Items” on page 37-8
- “External Mode Control Panel” on page 37-13
- “Target Interfacing” on page 37-15
- “External Signal Uploading and Triggering” on page 37-17
- “Data Archiving” on page 37-22
- “Parameter Downloading” on page 37-24

External Mode Interface Options

The ERT, GRT, GRT malloc, RSim, and Wind River Systems Tornado targets and the Real-Time Windows Target product support external mode. All targets that support it feature a set of external mode options on their respective target tab of the Configuration Parameters dialog box. This tab is normally named **Interface**). The next figure is from the GRT target dialog box, and is discussed below.



Note The xPC Target product also uses external mode communications. External mode in the xPC Target product is always on, and has no interface options.

The **Data exchange** section at the bottom has the following elements:

- **Interface** menu: Selects which of three mutually exclusive data interfaces to include in the generated code. Options are
 - None
 - C API
 - External mode
 - ASAP2

This chapter discusses only the **External mode** option. For information on other options, see “Specifying Target Interfaces” on page 33-16.

Once you select **External mode** from the Interface menu, the following options appear beneath:

- **Transport layer** menu: Identifies messaging protocol for host/target communications; choices are `tcpip` and `serial_win32`.

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; these will vary according to the protocol you use.

For more information on the transport options, see “Target Interfacing” on page 37-15 and “Client/Server Implementations” on page 37-35. You can add other transport protocols yourself by following instructions given in “Creating a TCP/IP Transport Layer for External Communication” on page 37-50.

- **Static memory allocation** check box: Controls how memory for external mode communication buffers in the target is allocated. When you select this option, the following one appears beneath it:
- **Static memory buffer size** text field: Number of bytes to preallocate for external mode communications buffers in the target when **Static memory allocation** is used.

Note Selecting **External mode** from the **Interface** menu does not cause the Simulink model to operate in external mode (see “External Mode Related Menu and Toolbar Items” on page 37-8, below). 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 Real-Time Workshop software to generate code for external mode that uses only static memory allocation (“malloc-free” code). When selected, it activates the **Static memory buffer size** edit field, in which you specify the size of the static memory buffer used by external mode. The default value is 1,000,000 bytes. Should 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.

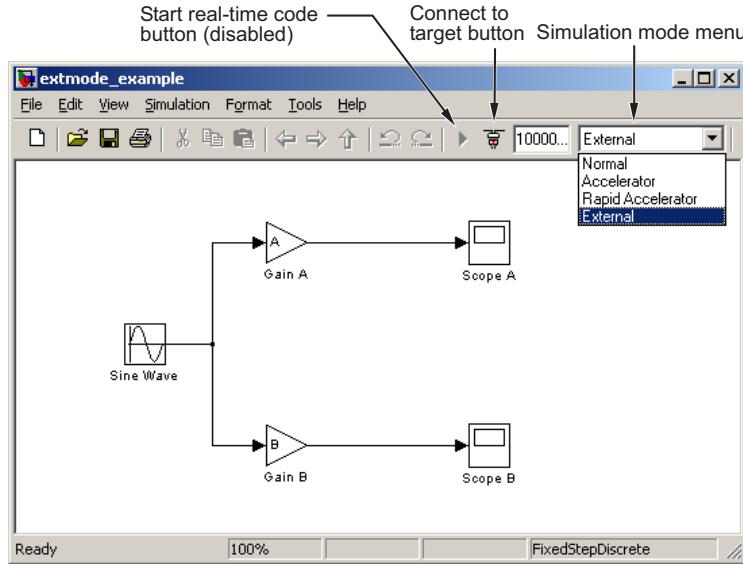
Notes

- To determine how much memory you need 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. Should an allocation fail, this console log can be used to adjust the size entered in the **Static memory buffer size** field.
 - When you create an ERT target, external mode can generate pure integer code. Select this feature by clearing the **Support floating-point numbers** option on the **Interface** pane of the Configuration Parameters dialog box or Model Explorer. Clearing this option ensures that all code, including external mode support code, is free of doubles and floats. For more details, see “Configuring the Target Hardware Environment” in the Real-Time Workshop Embedded Coder documentation.
-

External Mode Related Menu and Toolbar Items

To communicate with a target program, the model must be operating in external mode. To enable external mode, do either of the following:

- Select **External** from the **Simulation** menu.
- Select **External** from the simulation mode menu in the toolbar. The simulation mode menu is shown in the next figure.



Simulation Mode Menu Options and Target Connection Control (Host Disconnected from Target)

Once external mode is enabled, you can connect to and control the target program by doing any of the following:

- Select **Connect To Target** from the **Simulation** menu.
- Click the **Connect To Target** toolbar button, shown in the preceding figure.
- Use the **Ctrl+T** keyboard shortcut.

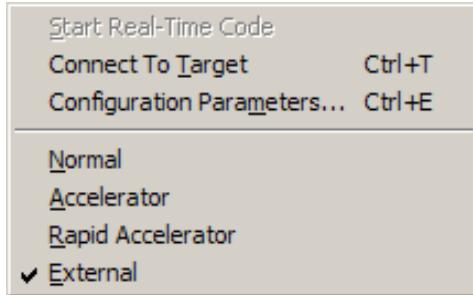
Note When external mode is selected in the model window, the **Ctrl+T** keyboard shortcut is remapped from a toggle for **Start** and **Stop** (simulation) to a toggle for **Connect To Target** and **Disconnect From Target**.

Selecting external mode in the model window controls execution only, and does *not* cause the Real-Time Workshop software to generate code for external mode. To do this, you must select **External mode** from the **Interface** menu on the **Interface** tab of the Configuration Parameters dialog box, as described in “External Mode Interface Options” on page 37-5.

Note You can enable external mode, and simultaneously connect to the target system, by using the External Mode Control Panel dialog box. See “External Mode Control Panel” on page 37-13.

Simulation Menu

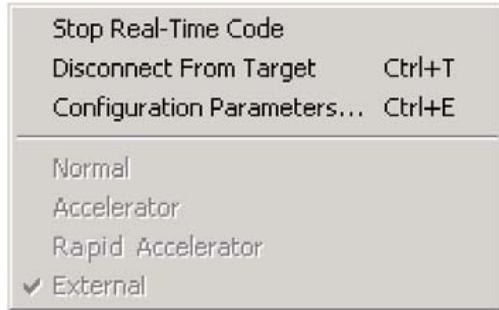
When a Simulink model is in external mode, the upper section of the **Simulation** menu contains external mode options. Initially, the Simulink model is disconnected from the target program, and the menu displays the options shown in the next figure.



Simulation Menu External Mode Options (Host Disconnected from Target)

The **Connect To Target** option establishes communication with the target program. When a connection is established, the target program might be executing model code, or it might be awaiting a command from the host to start executing model code. You can also accomplish this by clicking the **Connect To Target** toolbar button, as shown in Simulation Mode Menu Options and Target Connection Control (Host Disconnected from Target) on page 37-9.

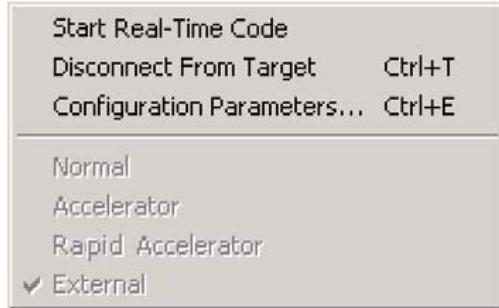
If the target program is executing model code, the **Simulation** menu contents change, as shown in the next figure.



Simulation Menu External Mode Options (Target Executing Model Code)

The **Disconnect From Target** option disconnects the Simulink model from the target program, which continues to run. The **Stop Real-Time Code** option terminates execution of the target program and disconnects the Simulink model from the target system.

If the target program is in a wait state, the **Start Real-Time Code** option is enabled, as shown in the next figure. The **Start Real-Time Code** option instructs the target program to begin executing the model code.



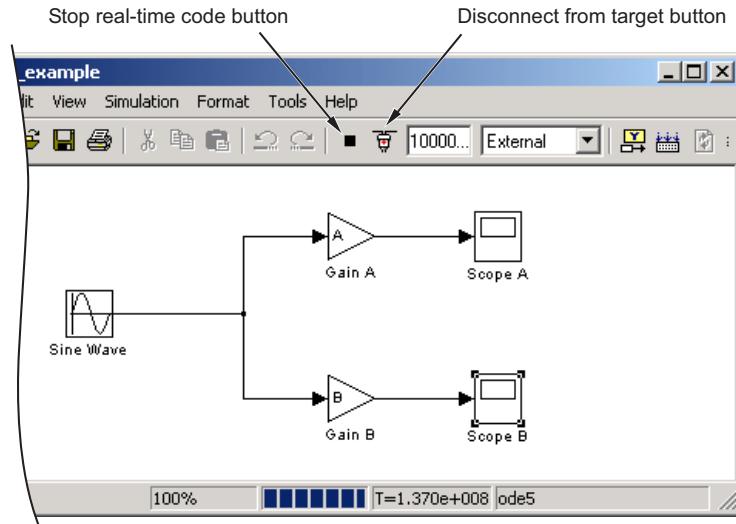
Simulation Menu External Mode Options (Target Awaiting Start Command)

Toolbar Controls

The Simulink toolbar controls, shown in [Simulation Mode Menu Options](#) and [Target Connection Control \(Host Disconnected from Target\) on page 37-9](#), let you control the same external mode functions as the **Simulation** menu. The Simulink model editor displays external mode buttons to the left of the Simulation mode menu. Initially, the toolbar displays a **Connect To Target** button and a disabled **Start real-time code** button. Click the **Connect To Target** button to connect the Simulink engine to the target program.

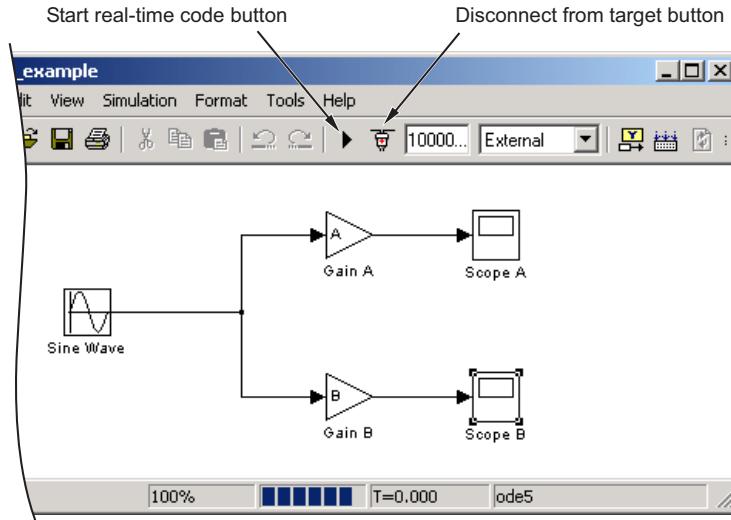
When a connection is established, the target program might be executing model code, or it might be awaiting a command from the host to start executing model code.

If the target program is executing model code, the toolbar displays a **Stop real-time code** button and a **Disconnect From Target** button (shown in External Mode Toolbar Controls (Target Executing Model Code) on page 37-12). Click the **Stop real-time code** button to command the target program to stop executing model code and disconnect the Simulink engine from the target system. Click the **Disconnect From Target** button to disconnect the Simulink engine from the target program while leaving the target program running.



External Mode Toolbar Controls (Target Executing Model Code)

If the target program is in a wait state, the toolbar displays a **Start real-time code** button and a **Disconnect From Target** button (shown in External Mode Toolbar Controls (Target in Wait State) on page 37-13). Click the **Start real-time code** button to instruct the target program to start executing model code. Click the **Disconnect From Target** button to disconnect the Simulink engine from the target program.



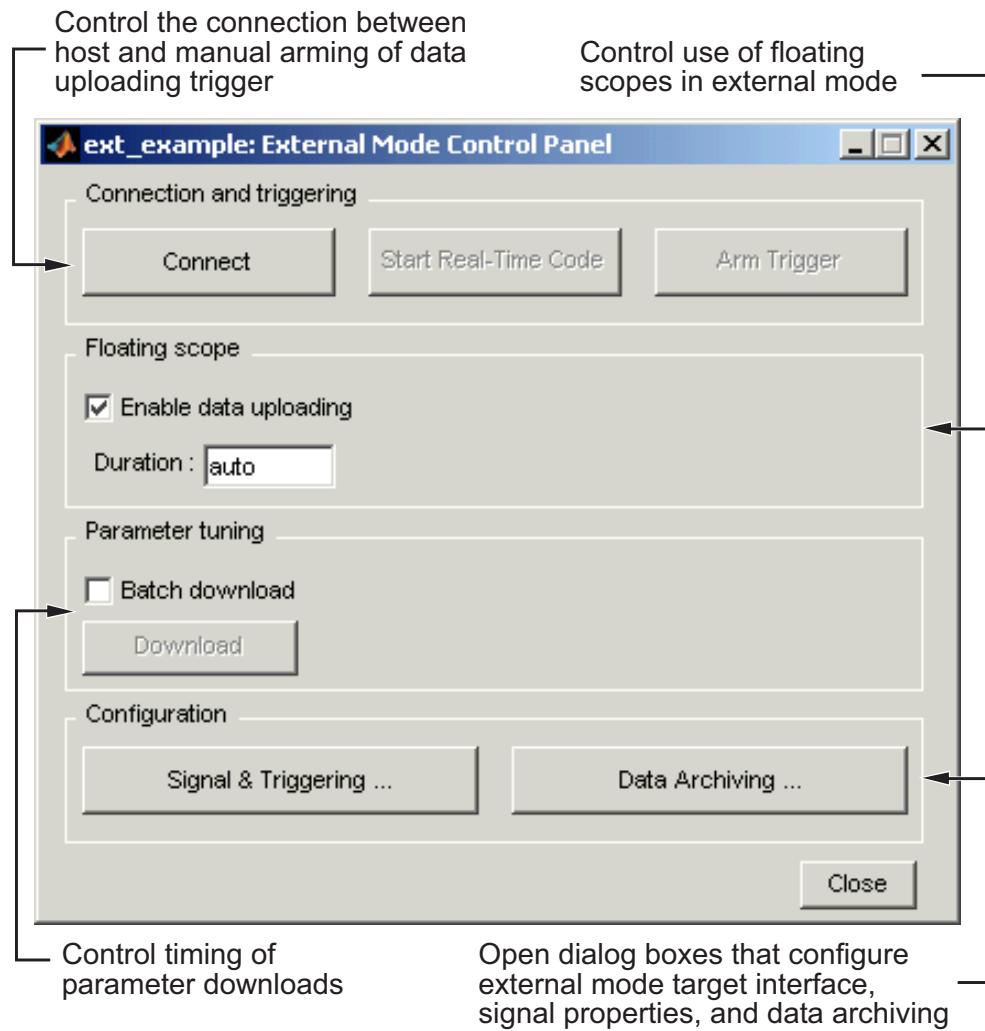
External Mode Toolbar Controls (Target in Wait State)

External Mode Control Panel

The External Mode Control Panel, illustrated in the next figure, provides centralized control of all external mode features, including

- Host/target connection, disconnection, and target program start/stop functions, and enabling of external mode
- Arming and disarming the data upload trigger
- External mode communications 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 features

Select **External Mode Control Panel** from the **Tools** menu on the Simulink model editor to open the External Mode Control Panel dialog box.



The following sections describe the features 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 “External Mode Related Menu and Toolbar Items” on page 37-8).

The **Connect/Disconnect** button connects to or disconnects from the target program. The button text changes in accordance with the state of the connection.

If external mode is not enabled at the time the **Connect** button is clicked, the External Mode Control Panel enables external mode automatically.

The **Start/Stop real-time code** button commands the target to start or terminate model code execution. The button is disabled until a connection to the target is established. The button text changes in accordance with the state of the target program.

Floating Scope Options

The **Floating scope** pane of the External Mode Control Panel controls when and for how long data is uploaded to Floating Scope blocks. When used under external mode, Floating Scopes

- Do not appear in the signal and triggering GUI
- Support manual triggering only

The behavior of wired scopes is not restricted in these ways.

The **Floating scope** pane contains a check box and an edit field:

- **Enable data uploading** check box, which functions as an **Arm trigger** button for floating scopes. When the target is disconnected it controls whether or not to arm when connecting the floating scopes. When already connected it acts as a toggle button to arm/cancel the trigger.
- **Duration** edit field, which specifies the duration for floating scopes. By default, it is set to `auto`, which causes whatever value is specified in the signal and triggering GUI (which by default is 1000 seconds) to be used.

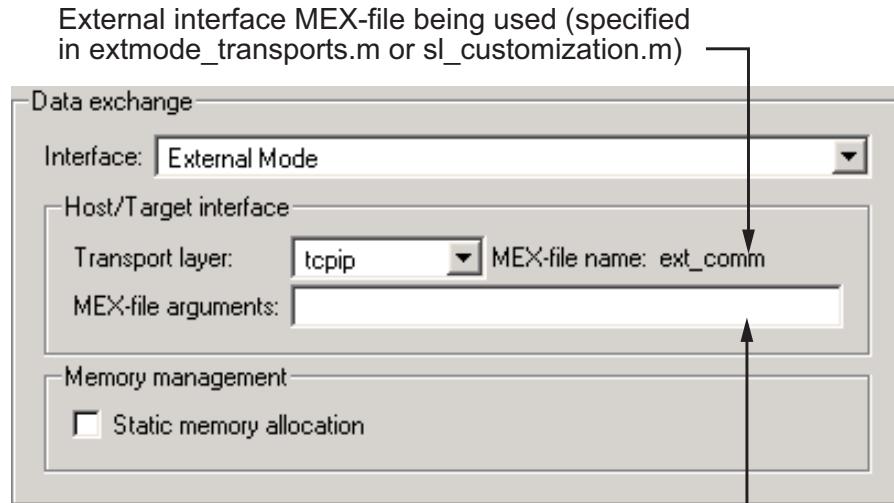
Target Interfacing

The Real-Time Workshop product lets you implement client and server transport for external mode using either TCP/IP or serial protocols. You can use the socket-based external mode implementation provided by the Real-Time Workshop product with the generated code, provided that your

target system supports TCP/IP. Otherwise, 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.

You specify the transport mechanism using the **Transport layer** menu in the **Host/Target interface** subpane of the **Interface** pane of the Configuration Parameters dialog box, shown below.



Type optional arguments to the external interface MEX-file here. This file must be in the current directory or one that is on your MATLAB path.

The **Host/Target interface** subpane also displays **MEX-file name**, the name of a MEX-file that implements host/target communications for the selected external mode transport layer. This is known as the external interface MEX-file. The default is `ext_comm`, the TCP/IP-based external interface file provided for use with the GRT, GRT malloc, ERT, RSim, and Tornado targets. If you select the `serial_win32` 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 on creating a custom transport layer, see “Creating a TCP/IP Transport Layer for External Communication” on page 37-50. For more information on specifying a TCP/IP or serial transport layer for a custom target, see “Using the TCP/IP Implementation” on page 37-35 or “Using the Serial Implementation” on page 37-38.

The **MEX-file arguments** edit field lets you 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)

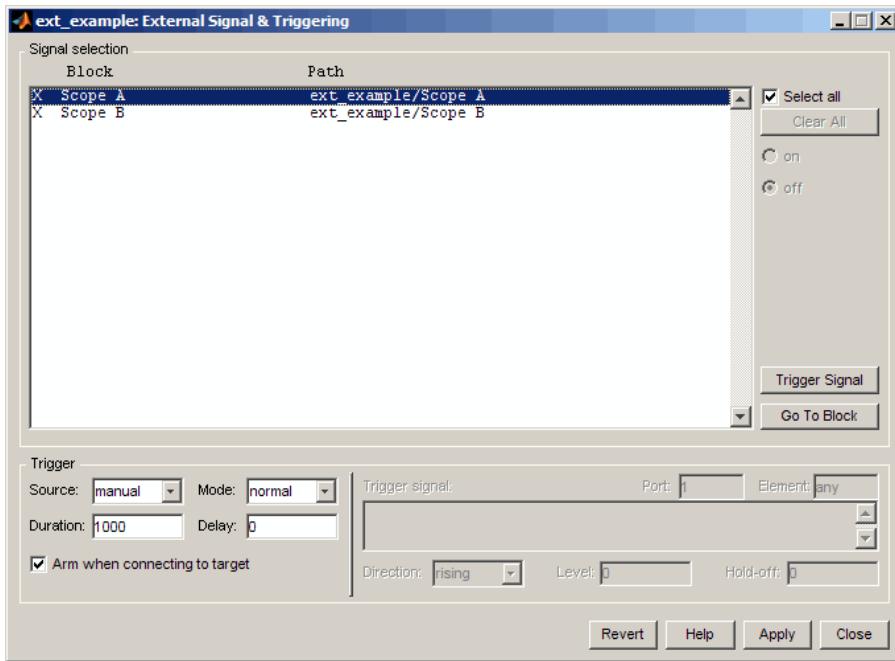
For serial transport, `ext_serial_win32_comm` allows three optional arguments:

- 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)

See “Client/Server Implementations” on page 37-35 for details on MEX-file transport architecture and arguments.

External Signal Uploading and Triggering

Clicking the **Signal & triggering** button of the External Mode Control Panel activates the External Signal & Triggering dialog box, as shown in the next figure.



The External Signal & Triggering dialog box displays a list of all blocks and subsystems in your model that support external mode signal uploading. See “External Mode Compatible Blocks and Subsystems” on page 37-27 for information on which types of blocks are external mode compatible.

The External Signal & Triggering dialog box lets you select the signals that are collected from the target system and viewed in external mode. It also lets you select a signal that triggers uploading of data when certain signal conditions are met, 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 is designed to simplify monitoring the target program. If you use the default settings, you do not need to preconfigure signals and triggers. Simply start the target program and connect the Simulink engine to it. All 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 is

- **Arm when connecting to target:** on
- **Trigger Mode:** normal
- **Trigger Source:** manual
- **Select all:** on

Signal Selection

All 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 to be viewed. An X appears to the left of each selected block's name.

The **Select all** check box selects all signals. By default, **Select all** is on.

If **Select all** is off, you can select or deselect individual signals using the **on** and **off** radio buttons. To select a signal, click the desired list entry and click the **on** radio button. To deselect a signal, click the desired list entry and click the **off** radio button. Alternatively, you can double-click a signal in the list to toggle between selection and deselection.

The **Clear all** button deselects all signals.

Trigger Options

The **Trigger** panel located at the bottom left of 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 start logging data when the **Arm trigger** button on the External Mode Control Panel is clicked.

Selecting **signal** tells external mode to start logging data when a selected trigger signal satisfies trigger conditions specified in the **Trigger signal** panel. When the trigger conditions are satisfied (that is, the signal crosses the trigger level in the specified direction) a *trigger event* occurs. If the

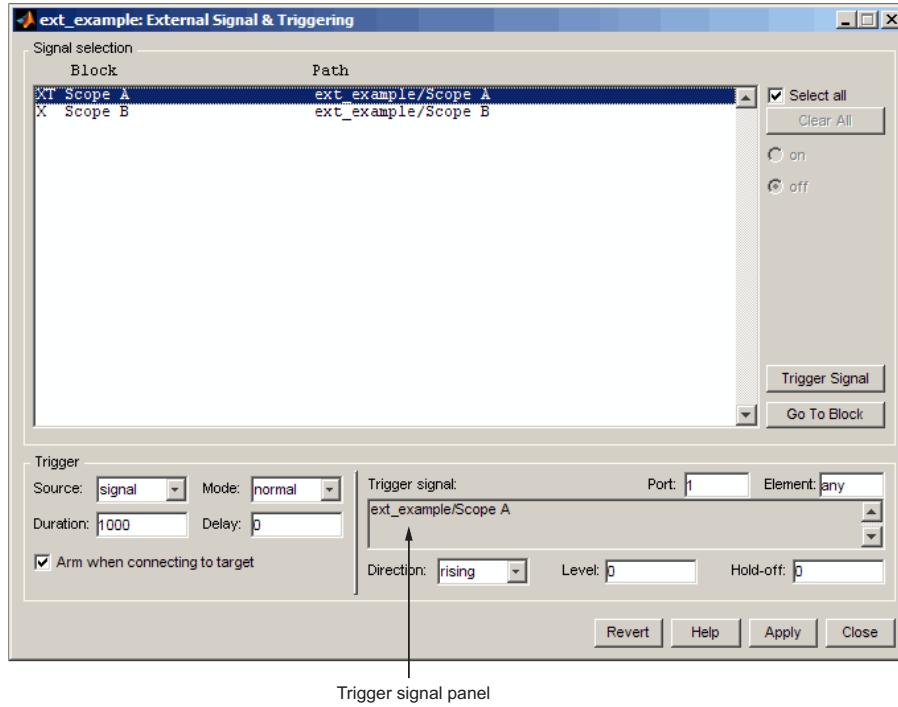
trigger is *armed*, external mode monitors for the occurrence of a trigger event. When a trigger event occurs, data logging begins.

- **Arm when connecting to target:** If this option is selected, external mode arms the trigger automatically when the Simulink engine connects to the target. If the trigger source is **manual**, uploading begins immediately. If the trigger mode is **signal**, monitoring of the trigger signal begins immediately, and uploading begins upon the occurrence of a trigger event.
If **Arm when connecting to target** is not selected, you must manually arm the trigger by clicking the **Arm trigger** button in the External Mode Control Panel.
- **Duration:** The number of base rate steps for which external mode logs data after a trigger event. For example, if the fastest rate in the model is 1 second and a signal sampled at 1 Hz is being logged for a duration of 10 seconds, then external mode will collect 10 samples. If a signal sampled at 2 Hz is logged, 20 samples will be collected.
- **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. See “Data Archiving” on page 37-22 for more details on the effect of the **Mode** setting.
- **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, and can be positive or negative. A negative delay corresponds to pretriggering. When the delay is negative, data from the time preceding the trigger is collected and uploaded.

Trigger Signal Selection

You can designate one signal as a trigger signal. To select a trigger signal, select **signal** from the **Trigger Source** menu. This activates the **Trigger signal** panel (see the next figure). Then, click the desired entry in the **Signal selection** list and click the **Trigger signal** button.

When a signal is selected as a trigger, a T appears to the left of the block’s name in 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 to the left of the block name.



External Signal & Triggering Window with Trigger Selected

After selecting the trigger signal, you can define the trigger conditions and set the **Port** and **Element** fields in the **Trigger signal** panel.

Setting Trigger Conditions

Note The **Trigger signal** panel and the **Port** and **Element** fields of the External Signal & Triggering dialog box are enabled only when trigger **Source** is set to **signal**.

By default, any element of the first input port of the 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** fields located on the right side of the **Trigger signal** panel. The **Port** field accepts a number or the keyword **last**. The **Element** field accepts a number or the keywords **any** and **last**.

The **Trigger Signal** panel defines the conditions under which a trigger event will occur.

- **Level:** Specifies a threshold value. The trigger signal must cross this value in a designated direction to fire the trigger. By default, the level is 0.
- **Direction:** `rising`, `falling`, or `either`. This specifies the direction in which the signal must be traveling when it crosses the threshold value. The default is `rising`.
- **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. Clicking the **Data Archiving** button of the External Mode Control Panel opens the External Data Archiving dialog box, which supports the following features.

Directory Notes

Use this option to add annotations that pertain to a collection of related data files in a directory. Clicking the **Edit directory note** button opens the MATLAB editor. Place comments that you want saved to a file in the specified directory in this window. By default, the comments are saved to the directory last written to by data archiving.

File Notes

Clicking **Edit file note** opens a file finder window that is, by default, set to the last file to which you have written. Selecting any MAT-file opens an edit window. Add or edit comments in this window that you want saved with your individual MAT-file.

Automated Data Archiving

Clicking the **Enable Archiving** check box activates the automated data archiving features of external mode. To understand how the archiving features work, it is necessary to consider the handling of data when archiving is not enabled. There are two cases, one-shot and normal mode.

In one-shot mode, after a trigger event occurs, each selected block writes its data to the workspace just 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*. Since the trigger can fire at any time, writing intermediate results to the workspace generally results 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 sufficient time exists between triggers for inspection of the intermediate results, then you can override the default behavior by checking the **Write intermediate results to workspace** check box. This option does not protect the workspace data from being overwritten by subsequent triggers.

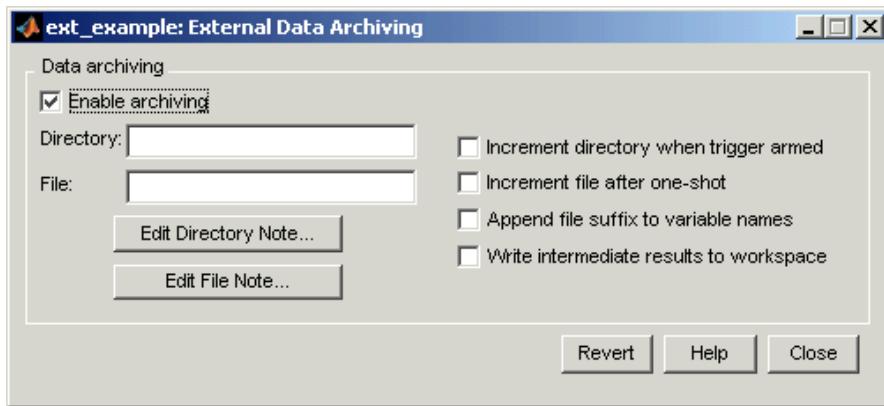
The options in the External Data Archiving dialog box support automatic writing of logging results, including intermediate results, to disk. Data archiving provides the following settings:

- **Directory:** Specifies the directory in which data is saved. External mode appends a suffix if you select **Increment directory when trigger armed**.
- **File:** Specifies the filename in which data is saved. External mode appends a suffix if you select **Increment file after one-shot**.
- **Increment directory when trigger armed:** External mode uses a different directory for writing log files each time that you click the **Arm trigger** button. The directories 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. This happens automatically in normal mode.
- **Append file suffix to variable names:** Whenever external mode increments filenames, 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 will save a variable named xdata in incremental files (file_1, file_2, and so on) as xdata_1, xdata_2, and so on. This is useful if you want to load the

MAT-files into the workspace and compare 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:** Select this option if you want the Real-Time Workshop software to write all intermediate results to the workspace.

The next figure shows the External Data Archiving dialog box with archiving enabled.



Unless you select **Enable archiving**, entries for the **Directory** and **File** fields are not accepted.

Parameter Downloading

The **Batch download** check box on the External Mode Control Panel enables or disables batch parameter changes.

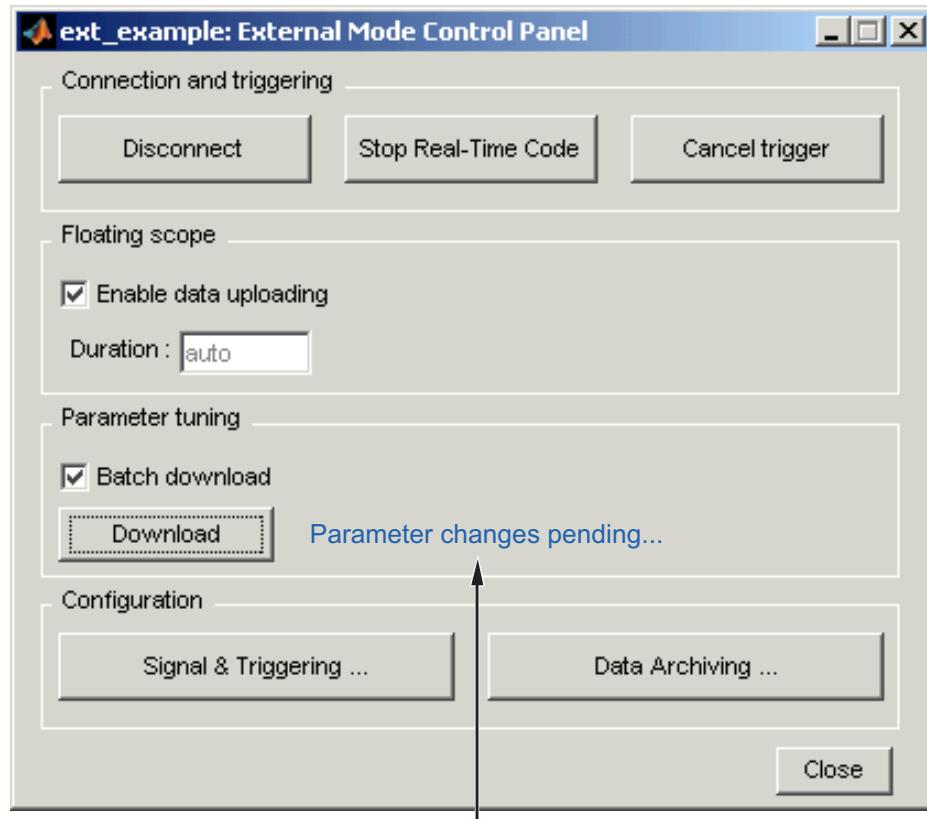
By default, batch download is not enabled. If batch download is not enabled, changes made directly to block parameters by using parameter dialog boxes are sent to the target when you click the **OK** or **Apply** button. Changes to MATLAB workspace variables are sent when an **Update diagram** is performed.

Note Opening a dialog box for a source block causes Simulink to pause. While Simulink is paused, you can edit the parameter values. You must close the dialog box to have the changes take effect and allow Simulink to continue.

If batch download is enabled, the Download button is enabled. Changes made to block parameters are stored locally until you click the Download button. When you click the Download button, the changes are sent in a single transmission.

When parameter changes have been made and are awaiting batch download, the External Mode Control Panel displays the message **Parameter changes pending...** to the right of the download button. (See the next figure.) This message disappears after the Simulink engine receives notification from the target that the new parameters have been installed in the parameter vector of the target system.

The External Mode Control Panel with the batch download option activated appears in the next figure.



Parameter changes pending...
message appears if unsent
parameter value changes
are awaiting download

External Mode Control Panel in Batch Download Mode

External Mode Compatible Blocks and Subsystems

In this section...

[“Compatible Blocks” on page 37-27](#)

[“Signal Viewing Subsystems” on page 37-28](#)

[“Supported Blocks for Data Archiving” on page 37-29](#)

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 and Vector Scope blocks from the Signal Processing Blockset 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 37-28 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, all 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, generating no code in the target system. Signal Viewing Subsystems run in all simulation modes — normal, accelerated, and external.

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.

See “Creating Subsystems” on page 3-37 for more information on atomic subsystems.

- 2 Use the following `set_param` command to turn the `SimViewingDevice` property on,

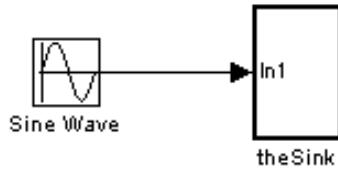
```
set_param('blockname', 'SimViewingDevice', 'on')
```

where '`blockname`' is the name of the subsystem.

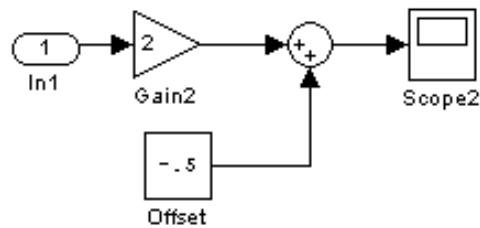
- 3 Make sure the subsystem meets the following requirements:

- It must be a pure sink block. That is, it must contain no 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 have no 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 their effect upon 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 External Data Archiving dialog box, as described in “Data Archiving” on page 37-22.

External Mode Communications Overview

In this section...

- “Introduction” on page 37-31
- “Download Mechanism” on page 37-31
- “Inlined and Tunable Parameters” on page 37-33

Introduction

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 37-31 describes the operation of external mode communications with **Inline parameters** off. “Inlined and Tunable Parameters” on page 37-33 describes the operation of external mode with **Inline parameters** on.

Download Mechanism

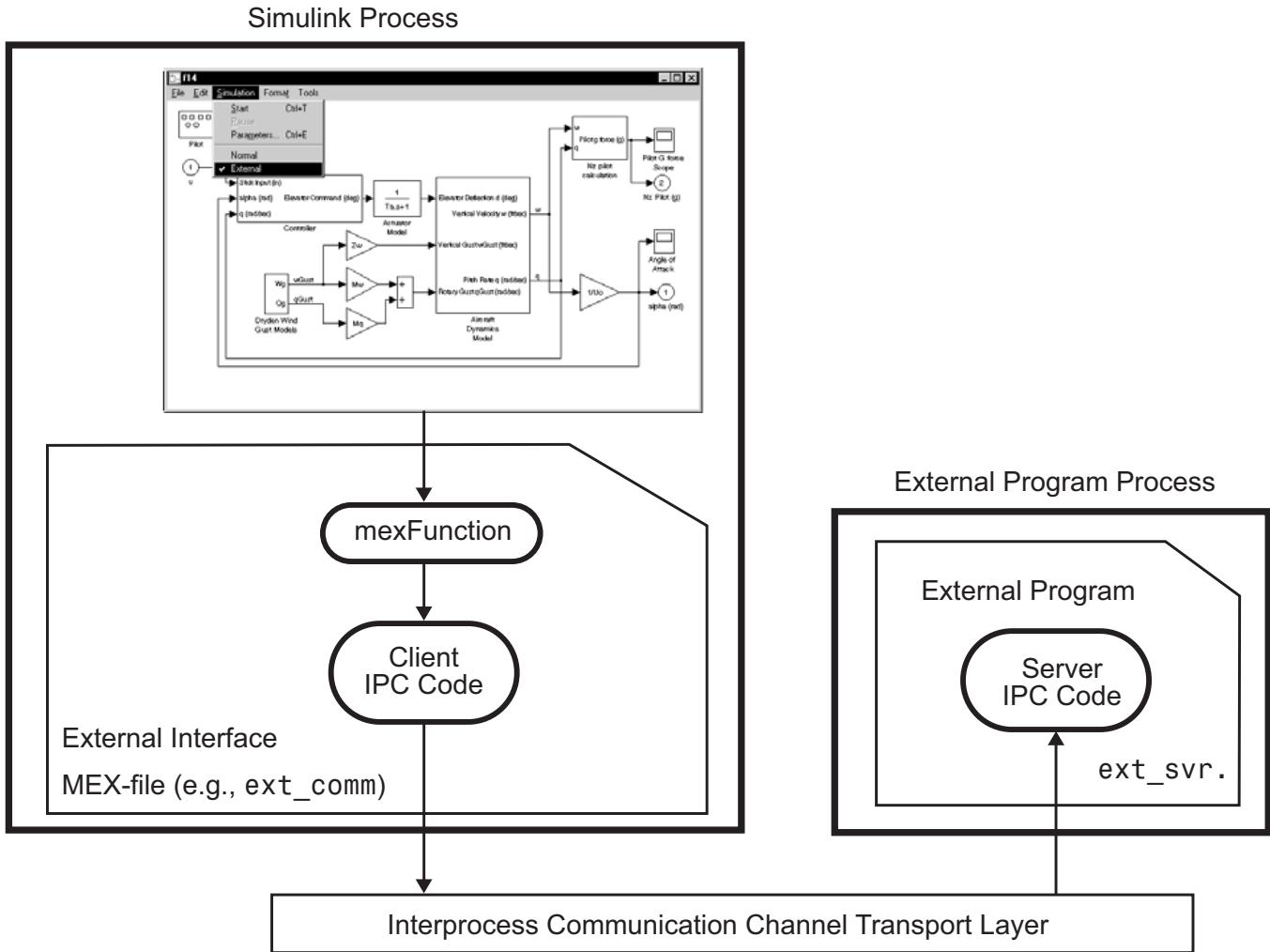
In external mode, the Simulink engine does not simulate the system represented by the block diagram. By default, when external mode is enabled, the Simulink engine downloads all parameters to the target system. After the initial download, the engine remains in a waiting mode until you change parameters 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.



External Mode Architecture

Inlined and Tunable Parameters

By default, all parameters (except those listed in “External Mode Limitations” on page 37-67) 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** pane of the Configuration Parameters dialog box), the Real-Time Workshop 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 Real-Time Workshop 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 Chapter 10, “Parameter Considerations”.

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 (if any) to the host. These values are assigned to the corresponding MATLAB workspace variables. This procedure ensures that the host and target are synchronized with respect to parameter values.

All 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 37-31 describes the operation of external mode communications with **Inline parameters** off.

Client/Server Implementations

In this section...

- “Introduction” on page 37-35
- “Using the TCP/IP Implementation” on page 37-35
- “Using the Serial Implementation” on page 37-38
- “Running the External Program” on page 37-41
- “Implementing an External Mode Protocol Layer” on page 37-43

Introduction

The Real-Time Workshop 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 Real-Time Workshop 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 37-15 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. Chapter 34, “Interfacing With a Real-Time Operating System” shows the use of external mode in the Wind River Systems Tornado environment. For help in customizing external mode transport layers, see “Creating a TCP/IP Transport Layer for External Communication” on page 37-50.

To use Simulink external mode over TCP/IP, you must

- Make sure that the correct external interface MEX-file is specified for your target's TCP/IP transport.

Targets provided by The 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`')

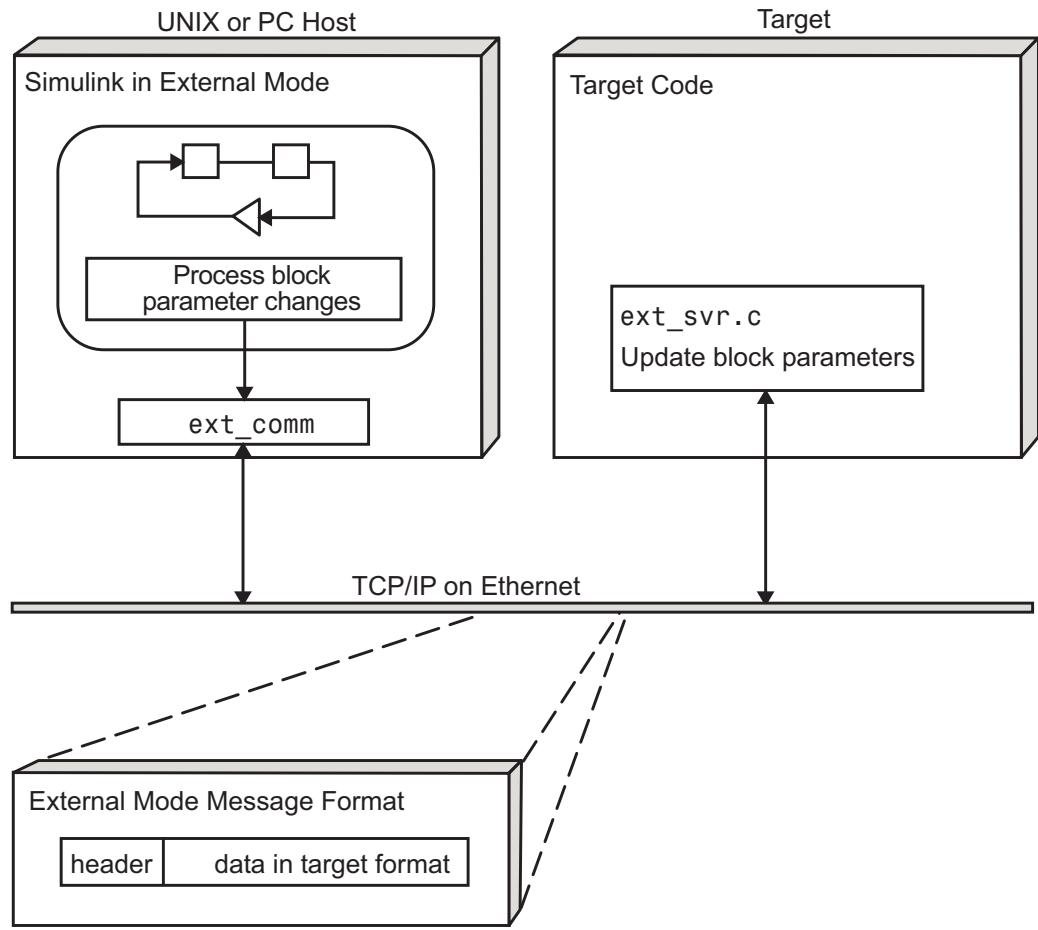
You can specify multiple targets and/or transports with additional `cm.ExtModeTransports.add` lines, for example:

```
function sl_customization(cm)
    cm.ExtModeTransports.add('mytarget.tlc', 'tcpip', 'ext_comm', 'Level1');
    cm.ExtModeTransports.add('mytarget.tlc', 'serial_win32', ...
                           'ext_serial_win32_comm', 'Level1');
%end function
```

- Be sure that the template makefile is configured to link the proper source files for the TCP/IP server code and that it defines the necessary 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 if necessary.

The arguments are positional and must be specified in order. 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 “Running the External Program” on page 37-41 for more information.

Using the Serial Implementation

Controlling host/target communications on a serial channel is similar to controlling host/target communications on a TCP/IP channel.

To use Simulink external mode over a serial channel, you must

- Execute the target and host on a Microsoft Windows platform.
- Make sure that the correct external interface MEX-file is specified for your target's serial transport.

Targets provided by The 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_win32`.

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_win32`')
- `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_win32', ...
                           'ext_serial_win32_comm', 'Level1');
%end function
```

- Be sure that the template makefile is configured to link the proper source files for the serial server code and that it defines the necessary 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

- 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
- Serial port ID (for example, 1 for COM1, and so on)

If the target program is executing on the same machine as the host and communications 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).

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 (selected from the set 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, with a default baud rate of 57600)

The arguments are positional and must be specified in order. 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 1 115200
```

You can specify command-line options to the external program when you launch it. The following section provides details on using command-line arguments.

Running the External Program

The external program must be running before you can use the Simulink product in external mode. 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 37-42.) 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 any options, as in the following example:

```
ext_example -tf inf -w
```

- Alternatively, you can launch the target executable from the MATLAB command prompt. In this case 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 any options, as in the following example:

```
ext_example -tf inf -w
```

- Alternatively, you can launch the target executable from the MATLAB command prompt. In the UNIX environment, if you start the external program from the MATLAB command prompt, 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 &
```

runs the executable from the MATLAB command prompt by spawning another process to run it.

Command-Line Options for the External Program

External mode target executables generated by the Real-Time Workshop code generator support the following command-line options:

- **-tf n** option

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. See “Using External Mode with the ERT Target” in the Real-Time Workshop Embedded Coder documentation.

Note The `-tf` option works with GRT, GRT malloc, ERT, RSim, and Tornado targets. If you are creating a custom target and want to support the `-tf` option, you must implement the option yourself. See “Creating a TCP/IP Transport Layer for External Communication” on page 37-50 for more information.

- `-w` option: 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` option: Specifies the TCP/IP port number, `n`, for the target program. The port number of the target program must match that of the host. The default port number is 17725. The port number must be a value between 256 and 65535.

Note The `-tf`, `-w`, and `-port` options are supported by the TCP/IP and serial transport layer modules shipped with the Real-Time Workshop product (although `-port` is interpreted differently by each). The `-baud` option is serial only. By default, these modules are linked into external mode target executables. If you are implementing a custom external mode transport layer and want to support these options, you must implement them in your code.

Implementing 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 Real-Time Workshop product and create a new external interface MEX-file. This advanced topic is described in detail in “Creating a TCP/IP Transport Layer for External Communication” on page 37-50.

Using 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 “Parameter Tuning by Using MATLAB Commands” on page 10-31.)

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	External Data Archiving: Append file suffix to variable names check box	Increment variable names for each incremented filename.
ExtModeArchiveDirName <i>string</i>	External Data Archiving: Directory text field	Save data in specified directory.
ExtModeArchiveFileName <i>string</i>	External Data Archiving: File text field	Save data in specified file.
ExtModeArchiveMode <i>string</i> - off , on	External 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 Real-Time Workshop software connects to the target.
ExtModeAutoIncOneShot off , on	External 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.

External Mode Command-Line Parameters (Continued)

Parameter and Values	Dialog Box Equivalent	Description
ExtModeChangesPending off , on	Not available	When ExtModeBatchMode is enabled, indicates whether any parameters remain in the queue of parameters to be downloaded to the target.
ExtModeCommand <i>string</i>	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.
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	External Data Archiving: Increment directory when trigger armed check box	Write log files to incremental directories each time the trigger is armed.
ExtModeLogAll off , on	External Signal & Triggering: Select all check box	Upload all available signals from the target to the host.
ExtModeLogCtrlPanelDlg <i>string</i>	Not available	Return a handle to the External Mode Control Panel dialog box or -1 if the dialog box does not exist.

External Mode Command-Line Parameters (Continued)

Parameter and Values	Dialog Box Equivalent	Description
ExtModeParamChangesPending off , on	Not available	When the Real-Time Workshop software is connected to the target and ExtModeBatchMode is enabled, indicates whether any 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 <i>integer (0)</i>	External Signal & Triggering: Delay text field	Specify the amount of time that elapses between a trigger occurrence and the start of data collection.
ExtModeTrigDirection <i>string - rising</i> , falling, either	External Signal & Triggering: Direction menu	Specify the direction in which the signal must be traveling when it crosses the threshold value.
ExtModeTrigDuration <i>integer (1000)</i>	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 <i>string - integer (auto)</i>	External Mode Control Panel: Duration text field	Specify the duration for floating scopes. If auto is specified, the value of ExtModeTrigDuration is used.

External Mode Command-Line Parameters (Continued)

Parameter and Values	Dialog Box Equivalent	Description
ExtModeTrigElement <i>string - integer, any, last</i>	External Signal & Triggering: Element text field	Specify the elements of the input port of the specified trigger block that can cause the trigger to fire.
ExtModeTrigHoldOff <i>integer (0)</i>	External Signal & Triggering: Hold-off text field	Specify the base rate steps between when a trigger event terminates and the trigger is rearmed.
ExtModeTrigLevel <i>integer (0)</i>	External Signal & Triggering: Level text field	Specify the threshold value the trigger signal must cross to fire the trigger.
ExtModeTrigMode <i>string - normal, oneshot</i>	External Signal & Triggering: Mode menu	Specify whether the trigger is to rearm automatically after each trigger event or whether only one buffer of data is to be collected each time the trigger is armed.
ExtModeTrigPort <i>string - integer (1), last</i>	External Signal & Triggering: Port text field	Specify the input port of the specified trigger block for which elements can cause the trigger to fire.
ExtModeTrigType <i>string - manual, signal</i>	External Signal & Triggering: Source menu	Specify whether to start logging data when the trigger is armed or when a specified trigger signal satisfies trigger conditions.

External Mode Command-Line Parameters (Continued)

Parameter and Values	Dialog Box Equivalent	Description
ExtModeUploadStatus <i>string</i> - inactive, armed, uploading	Not available	Return the status of the external mode upload mechanism — inactive, armed, or uploading.
ExtModeWriteAllDataToWs off , on	External Data Archiving: Write intermediate results to workspace check box	Write all intermediate results to the workspace.

Creating a TCP/IP Transport Layer for External Communication

In this section...

- “Introduction” on page 37-50
- “Design of External Mode” on page 37-50
- “External Mode Communications Overview” on page 37-54
- “External Mode Source Files” on page 37-55
- “Implementing a Custom Transport Layer” on page 37-59

Introduction

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 Real-Time Workshop programs, and with the basic operation of external mode. These topics are described in Chapter 3, “Architecture Considerations” and Chapter 37, “Communicating With Code Executing on a Target System Using Simulink 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, GRT malloc, 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. Serial transport is implemented only for Microsoft Windows 32-bit architectures.

The Real-Time Workshop product provides full source code for both the client and server-side external mode modules, as used by the GRT, GRT malloc, ERT, Rapid Simulation, and Tornado targets, and the Real-Time Windows Target and xPC Target products. The main client-side module is `ext_comm.c`. The main server-side module is `ext_svr.c`.

These two modules call the specified transport layer through the following source files.

Built-In Transport Layer Implementations

Protocol	Client or Server?	Source Files
TCP/IP	Client (host)	<ul style="list-style-type: none"> <code>matlabroot/rtw/ext_mode/common/rtiostream_interface.c</code> <code>matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/-rtiostream_tcpip.c</code>
	Server (target)	<ul style="list-style-type: none"> <code>matlabroot/rtw/c/src/ext_mode/common/rtiostream_interface.c</code> <code>matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/-rtiostream_tcpip.c</code>
Serial	Client (host)	<code>matlabroot/rtw/ext_mode/serial/ext_serial_transport.c</code>
	Server (target)	<code>matlabroot/rtw/c/src/ext_mode/serial/ext_svr_serial_transport.c</code>

For serial communication, the module `ext_serial_transport.c` implements the client-side transport functions and `ext_svr_serial_transport.c` contains 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/ext_mode/custom/ext_custom_transport.c`
(serial)

- Server (target) side:

`matlabroot/rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c`
(TCP/IP) or
`matlabroot/rtw/c/src/ext_mode/custom/ext_svr_custom_transport.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 Real-Time Workshop Embedded Coder documentation. Provided that you implement your `rtiostream` communications channel using the documented interface, it should not be necessary to change the file `rtiostream_interface.c` or any other external mode related files.

Note Do not modify working source files. Use the templates provided in the `/custom` or `/rtiostream` directory 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 Real-Time Workshop 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 “Creating a Custom Client (Host) Transport Protocol” on page 37-60.

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 appropriate targets.

For more details, see “Creating a Custom Server (Target) Transport Protocol for TCP/IP Communication” on page 37-64 or “Creating a Custom Server (Target) Transport Protocol for Serial Communication” on page 37-65.

External Mode Communications Overview

This section gives a high-level overview of how a Real-Time Workshop generated program communicates with Simulink external mode. This description is based on the TCP/IP version of external mode that ships with the Real-Time Workshop 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 37-56
- “Server (Target) Source Files” on page 37-57
- “Other Files in the Server Directory” on page 37-59

Client (Host) MEX-file Interface Source Files

The source files for the MEX-file interface component are located in the directory `matlabroot/rtw/ext_mode`, 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 Real-Time Workshop Embedded Coder documentation.

Provided that you implement your `rtiostream` communications channel using the documented interface, it should not be necessary to change the file `rtiostream_interface.c` or any 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 Real-Time Workshop software uses TCP/IP functions including `recv()`, `send()`, and `socket()`.

- `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 External Interfaces in the MATLAB online documentation 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, it might be necessary 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 Real-Time Workshop Embedded Coder documentation.

Provided that you implement your `rtiostream` communications channel using the documented interface, it should not be necessary to change the file `rtiostream_interface.c` or any 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 Real-Time Workshop software uses TCP/IP functions including `recv()`, `send()`, and `socket()`.

- `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 Directory

- `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`
- `serial/ext_serial_win32_port.c`

Implementing a Custom Transport Layer

- “Requirements” on page 37-60
- “Creating a Custom Client (Host) Transport Protocol” on page 37-60
- “Registering a Custom Client (Host) Transport Protocol” on page 37-62
- “Creating a Custom Server (Target) Transport Protocol for TCP/IP Communication” on page 37-64
- “Creating a Custom Server (Target) Transport Protocol for Serial Communication” on page 37-65

Requirements

- 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 appropriately.
- The target is assumed to support both `int32_T` and `uint32_T` data types.

Creating 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 `rtIOTStreamRecv` 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 Real-Time Workshop Embedded Coder documentation.

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 on page 37-61 for examples of `mex` invocations.

Replace the command-line entry `rtiostream_tcpip.c` with your custom component’s filename.

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 “Registering a Custom Client (Host) Transport Protocol” on page 37-62.

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
UNIX, TCP/IP	<pre>>> cd (matlabroot) >> mex rtw/ext_mode/common/ext_comm.c rtw/ext_mode/common/rtiostream_interface.c rtw/ext_mode/common/ext_convert.c rtw/c/src/rtiostream/rtiostreamtcpip/rtiostream_tcpip.c -Irtw/c/src -Irtw/c/src/ext_mode/common -Irtw/ext_mode/common -Irtw/c/src/rtiostream/utils -DSL_EXT_SO -ldl -output toolbox/rtw/rtw/ext_comm -lut</pre>
PC, TCP/IP	<pre>>> cd (matlabroot) >> mex rtw\ext_mode\common\ext_comm.c rtw\ext_mode\common\ext_convert.c rtw\ext_mode\common\rtiostream_interface.c rtw\c\src\rtiostream\rtiostreamtcpip\rtiostream_tcpip.c -Irtw\c\src\ext_mode\common -Irtw\c\src -Irtw\ext_mode\common -Irtw\c\src\rtiostream\utils -DSL_EXT_DLL -DWIN32 wsock32.lib -output toolbox\rtw\rtw\ext_comm -Lextern\lib\win32\microsoft -lut</pre> <p>Note The argument <code>-Lextern\lib\win32\microsoft</code> is compiler specific. For example, if you are using the LCC compiler, you must replace this with <code>-Lextern\lib\win32\lcc</code>.</p>

MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files (Continued)

Platform	Commands
PC, serial	<pre>>> cd (matlabroot) >> mex rtw\ext_mode\common\ext_comm.c rtw\ext_mode\common\ext_convert.c rtw\ext_mode\serial\ext_serial_transport.c rtw\c\src\ext_mode\serial\ext_serial_pkt.c rtw\c\src\ext_mode\serial\ext_serial_win32_port.c -Irtw\c\src\ext_mode\common -Irtw\c\src\ext_mode\serial -Irtw\ext_mode\common -output toolbox\rtw\rtw\ext_serial_win32_comm -DWIN32</pre>

Note mex requires a compiler supported by the MATLAB API. See the `mex` reference page and External Interfaces in the MATLAB online documentation for more information about the `mex` function.

Registering 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 function
```

where

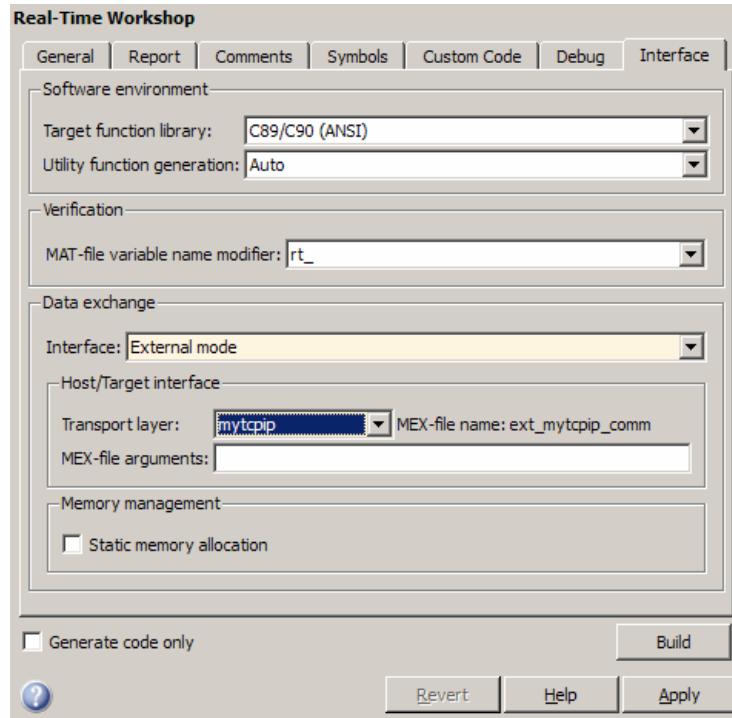
- `stf.tlc` is the name of the system target file for which the transport will be registered (for example, `'grt.tlc'`)
- `transport` is the transport name to display in the **Transport layer** menu on the **Interface** pane of the Configuration Parameters dialog box (for example, `'mytcpip'`)

- *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 function
```

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. Assuming an appropriate system target file is selected for your model, 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.



Creating a Custom Server (Target) Transport Protocol for TCP/IP Communication

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`.

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`; however, in most cases you will not need to modify `rtiostream_interface.c` for your custom TCP/IP transport layer.

To implement the server (target) side of your low-level TCP/IP transport protocol,

1 Edit the template

`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 drivers.

You must implement all 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` for guidance as needed.

- 2** Modify all appropriate template makefiles to support the new transport. 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`, and other server code into your executable.

Creating a Custom Server (Target) Transport Protocol for Serial Communication

The `Ext*` function prototypes in
`matlabroot/rtw/c/src/ext_mode/common/ext_svr_transport.h`
define the calling interface for the server (target) side
transport layer functions. The serial implementations are in
`matlabroot/rtw/c/src/ext_mode/serial/ext_svr_serial_transport.c`.

To implement the server (target) side of your low-level serial transport protocol,

1 Edit the template

`matlabroot/rtw/c/src/ext_mode/custom/ext_svr_custom_transport.c`
to replace low-level communication calls with your own
communication calls.

- a** Copy and rename the file to `ext_svr_name_transport.c` (replacing `name` with a name meaningful to you).
- b** Replace the functions in the **VISIBLE FUNCTIONS** section with functions that call your low-level communication primitives. These are the functions called from other target modules such as the main program.

You must implement all the functions defined in `ext_svr_transport.h`, and your implementations must conform to the prototypes defined in that file. Refer to `ext_svr_serial_transport.c` for guidance as needed.

- c** Supply a definition for the `ExtUserData` structure. This structure is required. If `ExtUserData` is not necessary for your external mode implementation, define an `ExtUserData` structure with one dummy field.
- d** Define the `EXT_BLOCKING` conditional as appropriate for your implementation:
 - Define `EXT_BLOCKING` as 0 to poll for a connection to the host (appropriate for single-threaded applications).
 - Define `EXT_BLOCKING` as 1 in multithreaded applications where tasks are able to block for a connection to the host without blocking the entire program.

See also the comments on `EXT_BLOCKING` in
`ext_svr_custom_transport.c`.

The `ext_svr_*_transport` source code modules are fully commented. See those files for more details.

- 2** If you created an `ext_name_utils.c` file to define custom transport symbols and functions (see step 2 of “Creating a Custom Client (Host) Transport Protocol” on page 37-60), and if the file is needed for your server side protocol, include it in your custom server transport source file.
- 3** Modify all appropriate template makefiles to support the new transport. 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 `ext_svr_name_transport.c` and other server code into your executable.

External Mode Limitations

In this section...

[“Limitations on Changing Parameters” on page 37-67](#)

[“Limitation on Mixing 32-bit and 64-bit Architectures” on page 37-68](#)

[“Limitations on Uploading Data” on page 37-68](#)

[“Limitations on Archiving Data” on page 37-68](#)

Limitations 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 any block
- The sample time or the number of sample times
- The integration algorithm for continuous systems
- The name of the model or of any block
- The parameters to the Fcn block

If you cause any of 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.

Note Opening a dialog box for a source block causes Simulink to pause. While Simulink is paused, you can edit the parameter values. You must close the dialog box to have the changes take effect and allow Simulink to continue.

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 the correct 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 Real-Time Workshop 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.

Limitations on Uploading Data

External mode does not support uploading data values for fixed-point or enumerated types into workspace parameters.

Limitations 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.

Accessing Memory in Generated Code

The Real-Time Workshop 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 all 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 the Target Language Compiler documentation 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.

Setting Up Runtime Logging to MAT-Files

- “About Logging to MAT-Files” on page 39-2
- “Configuring a Model to Log States, Time, and Output” on page 39-3
- “Logging Data with Scope and To Workspace Blocks” on page 39-5
- “Logging Data with To File Blocks” on page 39-6
- “Data Logging Differences in Single- and Multitasking Models” on page 39-7

About Logging to MAT-Files

Multiple techniques are available by which a program generated by the Real-Time Workshop software can save data to a MAT-file for analysis. See also “Data Logging” in Getting Started for a tutorial on data logging features.

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.

Configuring a Model to Log States, Time, and Output

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 Exporting Data to the MATLAB Workspace in the Simulink documentation.

For each workspace return variable that you define and enable, the Real-Time Workshop 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 Real-Time Workshop code generator logs the following data:

- All 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.

- All 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`.

In this section...

[“Overriding the Default MAT-File Name” on page 39-3](#)

[“Overriding the Default MAT-File Variable Names” on page 39-4](#)

Overriding the Default MAT-File Name

The MAT-file name defaults to `model.mat`. To specify a different filename,

- 1** In the model window, select **Simulation > Configuration Parameters**.
The dialog box opens.
- 2** Click **Real-Time Workshop**.
- 3** Append the following option to the existing text in the **Make command** field.

```
OPTS= " -DSAVEFILE=filename "
```

Overriding the Default MAT-File Variable Names

By default, the Real-Time Workshop 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,

- 1** In the model window, select **Simulation > Configuration Parameters**.
The dialog box opens.
- 2** Click **Real-Time Workshop**.
- 3** Select `grt.tlc` for **System target file**.
- 4** Select **Real-Time Workshop > Interface**
- 5** Select a prefix (`rt_`) or suffix (`_rt`) from the **MAT-file variable name modifier** field, or choose `none` for no prefix (other targets may or may not have this option).

Logging Data with Scope and To Workspace Blocks

The code generated by the Real-Time Workshop code generator also logs data from these sources:

- All 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.
- All 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 any variables logged from the Workspace I/O pane.

Logging 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's time and input variable(s). You must specify the filename, variable names, decimation, and sample time in the To File block's 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 Real-Time Workshop software for referenced models does not perform data logging to MAT-files.

Data Logging Differences in Single- and Multitasking Models

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 Real-Time Workshop software 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 Chapter 42, “Testing and Refining Concept Models With Standalone Rapid Simulations” for more information.

Sharing Utility Functions

- “About Shared Utility Functions” on page 40-2
- “Controlling Shared Utility Function Generation” on page 40-3
- “rtwtypes.h and Shared Utility Functions” on page 40-4
- “Incremental Shared Utility Function Generation and Compilation” on page 40-4
- “Shared Utility Checksum” on page 40-6
- “Shared Fixed-Point Utility Functions” on page 40-8

About Shared Utility Functions

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 functionality for each block instance.

Typically, functions that can have multiple callers are packaged into a library. Traditionally, such functions are defined statically, that is, the function source code exists in a file before you use the Real-Time Workshop software to generate code for your model.

In other cases, several model- and block-specific properties can affect which functions are needed and their behavior. Additionally, these properties can affect 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 Real-Time Workshop shared utility mechanism, which generates any needed support functions during code generation process.

Controlling Shared Utility Function Generation

You control the shared utility function generation mechanism with the **Utility function generation** option on the **Real-Time Workshop > 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 any Model blocks, the Real-Time Workshop build process places any 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 Real-Time Workshop build directory (for example, *vdp_grt_rtw*)

Thus, the code is specific to the model.

If a model does contain Model blocks, the Real-Time Workshop build process creates and uses a shared utilities directory within *slprj*. Model reference builds require the use of shared utilities. The naming convention for shared utility directories is *slprj/target/_sharedutils*, where *target* is *sim* for simulations with Model blocks or the name of the system target file for Real-Time Workshop target builds. Some examples follow:

```
slprj/sim/_sharedutils      % directory used with simulation
slprj/grt/_sharedutils      % directory used with grt.tlc STF
slprj/ert/_sharedutils      % directory used with ert.tlc STF
slprj/mytarget/_sharedutils % directory used with mytarget.tlc STF
```

To force a model build to use the *slprj* directory for shared utility generation, even when the current model contains no Model blocks, set the **Utility function generation** option to **Shared location**. This forces the Real-Time Workshop build process to place utilities under the *slprj* directory rather than in the normal Real-Time Workshop build directory. 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 Functions

The generated header file `rtwtypes.h` provides necessary defines, enumerations, and so on. The location of this file is controlled by whether the build process is using the shared utilities directory. Typically, the Real-Time Workshop build process places `rtwtypes.h` in the standard build directory, `model_target_rtw`. However, if a shared directory is required, the product places `rtwtypes.h` in `s1prj/target/_sharedutils`.

Incremental Shared Utility Function Generation and Compilation

As explained in “Controlling Shared Utility Function Generation” on page 40-3, you can specify that C source files, which contain function definitions, and header files, which contain macro definitions, be generated in a shared utilities directory. For the purpose of this discussion, the term functions means functions and macros.

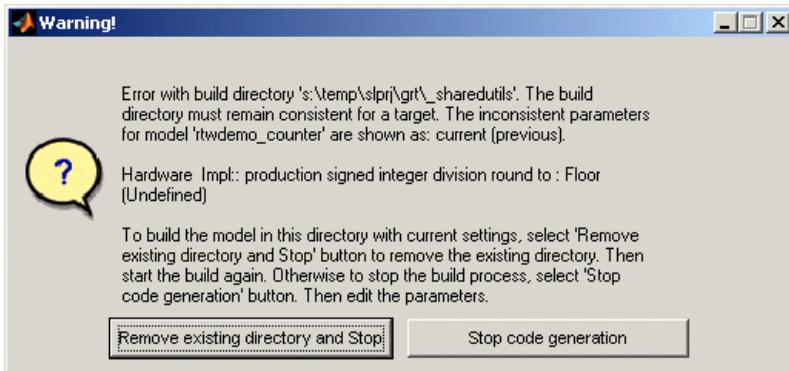
A shared function can be used by blocks within the same model and by blocks in different models when using model reference or when building multiple standalone models from the same start build directory. However, the Real-Time Workshop software generates the code for a given function only once for the 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 function 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 affect function behavior are included in a shared utility checksum or affect the function and file name.
- Block properties that affect the function behavior also affect the function and file name.

During compilation, makefile rules for the shared utilities directory 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 Function Generation and Compilation” on page 40-4, the Real-Time Workshop software uses the shared utilities directory 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 directory, the configuration file `slprj/target/_sharedutils/checksummap.mat` exists, the product reads that file and ensures 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	<pre>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')</pre>

Category	Properties
	<pre>get_param(bdroot, 'ProdWordSize') get_param(bdroot, 'TargetHWDeviceType') get_param(bdroot, 'ProdHWDeviceType') get_param(bdroot, 'TargetIntDivRoundTo') get_param(bdroot, 'ProdIntDivRoundTo')</pre>
Additional configuration properties	<pre>get_param(bdroot, 'TargetLibSuffix') get_param(bdroot, 'TargetLang') get_param(bdroot, 'TemplateMakefile')</pre>
ERT target properties	<pre>get_param(bdroot, 'PurelyIntegerCode') get_param(bdroot, 'SupportNonInlinedSFcns')</pre>
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 directory 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 ensures that several critical properties are identical for all models that use the shared utilities. For the fixed-point functions, there are additional properties that affect function behavior. These properties are coded into the functions and file names to ensure requirements are maintained. The additional properties include

Category	Function/Property
Block properties	<ul style="list-style-type: none"> • Fixed-point operation being performed by the block • Fixed-point data type and scaling (Slope, Bias) of function inputs and outputs • Overflow handling mode (Saturation, Wrap) • Rounding Mode (Floor, Ceil, Zero)
Model properties	<code>get_param(bdroot, 'NoFixptDivByZeroProtection')</code>

The naming convention for the fixed-point utilities is based on the properties as follows:

```
operation + [zero protection] + output data type + output bits +
[input1 data] + input1 bits + [input2 data type + input2 bits] +
[shift direction] + [saturate mode] + [round mode]
```

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
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.

Verifying Generated Code Applications

- Chapter 41, “Testing and Refining Concept Models With Host-Based Simulations”
- Chapter 42, “Testing and Refining Concept Models With Standalone Rapid Simulations”
- Chapter 43, “Verifying Generated Code”
- Chapter 44, “Testing and Refining a Model With Rapid Prototyping”
- Chapter 45, “Verifying Generated Source Code With Software-In-the-Loop Simulation”
- Chapter 46, “Verifying a System With Hardware-In-the-Loop Simulation”



Testing and Refining Concept Models With Host-Based Simulations

- “About Host-Based Simulation” on page 41-2
- “Host-Based Simulation Performance” on page 41-3
- “Goals of Host-Based Simulation” on page 41-4
- “Running Host-Based Simulations” on page 41-5

About Host-Based Simulation

You can apply Real-Time Workshop technology early on in the design process by validating requirements with host-based simulations. A host computer runs MATLAB and Simulink in a non-real-time (general purpose) operating system environment. Non-real-time operating systems are nondeterministic. For example, they might suspend code execution to run an operating system service and then, after providing the service, continue executing your application code. The executable for your generated code might run faster or slower than the sample rates that you specified in your model.

Using Simulink, you can create, simulate, analyze, and maintain models for components (algorithms and controllers) and the operating environments (or plants) that they control, throughout a product life cycle.

Host-Based Simulation Performance

As your models and simulation scenarios increase in size and complexity, you can adjust Simulink simulation modes to improve simulation performance. Normal simulations run in an interpretive mode that allows you to access, display, and tune data and parameters. This mode is useful for experimentation and initial model development. If normal mode simulations are not fast enough, you can improve simulation performance by using alternative modes, Accelerator mode and Rapid Accelerator mode, which leverage Real-Time Workshop technology by replacing the interpreted code with compiled target code.

Note Code generated by the Accelerator and Rapid Accelerator modes is suitable only for speeding up model simulation. You must use the Real-Time Workshop product to generate code for other purposes.

Goals of Host-Based Simulation

Assuming that you have documented application requirements and developed a concept model, you can use host-based simulations to:

- Validate functionality of a model in non-real time
- Refine a concept model for an algorithm
- Test a concept model

Running Host-Based Simulations

For information on how to run and speed up host-based simulations, see the following topics in the Simulink documentation:

- “Running Simulations”
- “Accelerating Models”

Testing and Refining Concept Models With Standalone Rapid Simulations

- “About Rapid Simulation” on page 42-2
- “Rapid Simulation Performance” on page 42-3
- “General Rapid Simulation Workflow” on page 42-4
- “Identifying Your Rapid Simulation Requirements” on page 42-6
- “Configuring Import Blocks to Provide Rapid Simulation Source Data” on page 42-8
- “Configuring and Building a Model for Rapid Simulation” on page 42-9
- “Setting Up Rapid Simulation Input Data” on page 42-12
- “Programming Scripts for Batch and Monte Carlo Simulations” on page 42-23
- “Running Rapid Simulations” on page 42-24
- “Rapid Simulation Target Limitations” on page 42-37

About Rapid Simulation

After you create a model, you can use the Real-Time Workshop rapid simulation (RSim) target to characterize the model's 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.

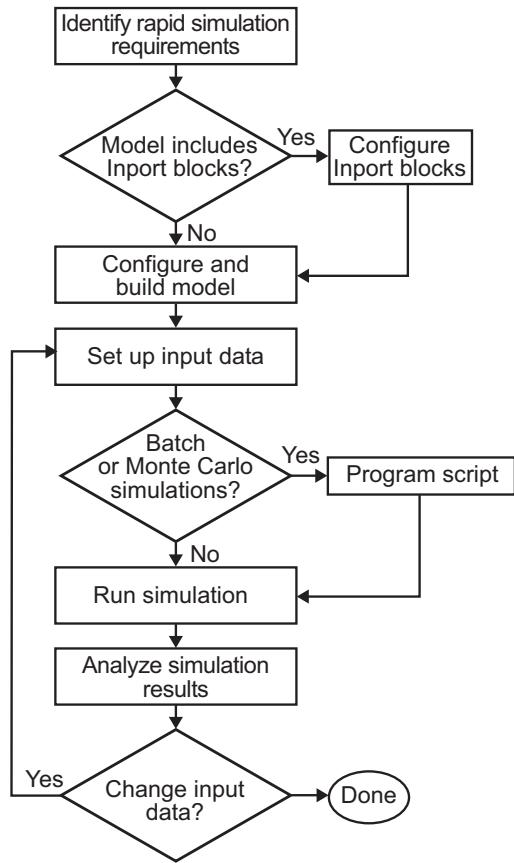
Rapid Simulation Performance

The performance 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 any 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.

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 Import 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.



Identifying Your 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?	“Configuring and Building a Model for Rapid Simulation” on page 42-9
Are there any solver requirements? Do you expect to use the same solver for which the model is configured for your rapid simulations?	“Configuring and Building a Model for Rapid Simulation” on page 42-9
Do your rapid simulations need to accommodate flexible custom code interfacing? Or, do your simulations need to retain storage class settings?	“Configuring and Building a Model for Rapid Simulation” on page 42-9
Will you be running simulations with multiple data sets?	“Setting Up Rapid Simulation Input Data” on page 42-12
Will the input data consist of global parameters, signals, or both?	“Setting Up Rapid Simulation Input Data” on page 42-12
What type of source blocks provide input data to the model — From File, Import, From Workspace?	“Setting Up Rapid Simulation Input Data” on page 42-12
Will the model’s parameter vector (<i>model_P</i>) be used as input data?	“Creating a MAT-File That Includes a Model Parameter Structure” on page 42-13
What is the data type of the input parameters and signals?	“Setting Up Rapid Simulation Input Data” on page 42-12
Will the source data consist of one variable or multiple variables?	“Setting Up Rapid Simulation Input Data” on page 42-12
Does the input data include tunable parameters?	“Creating a MAT-File That Includes a Model Parameter Structure” on page 42-13

Question...	For More Information, See...
Do you need to gain access to tunable parameter information — model checksum and parameter data types, identifiers, and complexity?	“Creating a MAT-File That Includes a Model Parameter Structure” on page 42-13
Will you have a need to vary the simulation stop time for simulation runs?	“Configuring and Building a Model for Rapid Simulation” on page 42-9 and “Overriding a Model Simulation Stop Time” on page 42-27
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.	“Setting a Clock Time Limit for a Rapid Simulation” on page 42-26
Do you need to preserve the output of each simulation run?	“Specifying a New Output File Name for a Simulation” on page 42-36 and “Specifying New Output File Names for To File Blocks” on page 42-36
Do you expect to run the simulations interactively or in batch mode?	“Programming Scripts for Batch and Monte Carlo Simulations” on page 42-23

Configuring Import Blocks to Provide Rapid Simulation Source Data

You can use Import 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 Import 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**

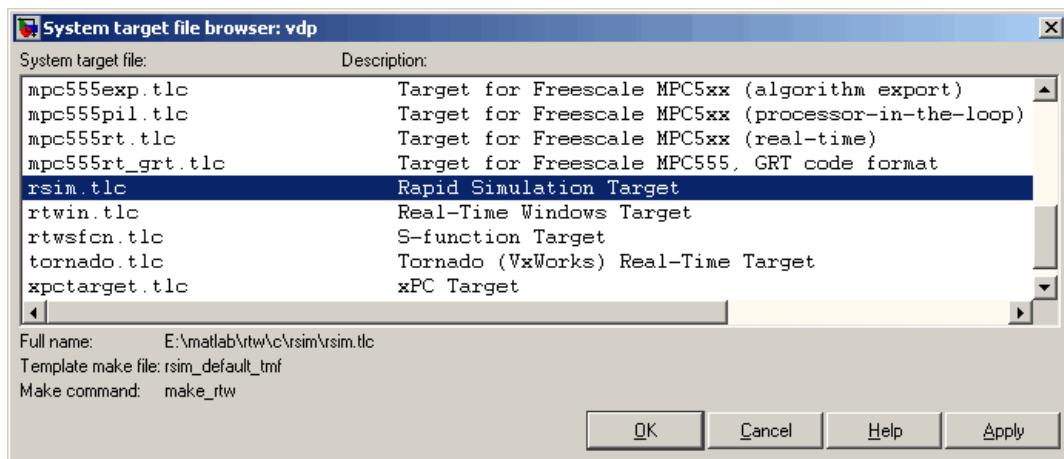
If you do not have control over the model content, you might need to modify the data in the MAT-file to conform to what the model expects for input. Input data characteristics and specifications of the Import block that receives the data must match.

For details on adjusting these parameters and on creating a MAT-file for use with an Import block, see “Creating a MAT-File for an Import Block” on page 42-18. For descriptions of the preceding block parameters, see the description of the Import block in the Simulink documentation.

Configuring and Building a Model for Rapid Simulation

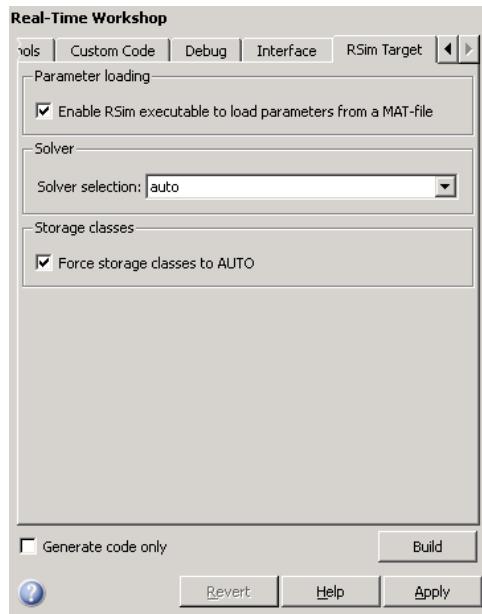
After you identify your rapid simulation requirements, configure the model for rapid simulation.

- 1** Open the Configuration Parameters dialog box.
- 2** Click **Real-Time Workshop**.
- 3** On the **Real-Time Workshop** pane, click **Browse**. The System Target File Browser opens.
- 4** Select **rsim.tlc** (Rapid Simulation Target) and click **OK**.



On the **Real-Time Workshop** pane, the Real-Time Workshop software populates the **Make command** and **Template makefile** fields with default settings and adds the **RSim Target** tab or node under **Real-Time Workshop**.

- 5** Click **RSim Target** to view the **RSim Target** pane.



- 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 Real-Time Workshop 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 RTW fixed-step solvers . In the Configuration Parameters dialog box, on the Solver pane, specify a fixed-step solver.

If You Want to...	Then...
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 all storage classes to Auto for flexible custom code interfacing	Select Force storage classes to AUTO (default).
Retain storage class settings, such as <code>ExportedGlobal</code> or <code>ImportedExtern</code> , 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 needed. By default, the Real-Time Workshop software saves simulation logging results to a file named *model.mat*. For more information, see “Importing and Exporting Data” in the Simulink documentation.
- 8 If appropriate for your simulations, set up external mode communications on the **Real-Time Workshop > Interface** pane. See Chapter 37, “Communicating With Code Executing on a Target System Using Simulink External Mode” for details.
- 9 Return to the **Real-Time Workshop** pane and click **Build**. The Real-Time Workshop 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 Real-Time Workshop product, see “Choosing and Configuring a Compiler” on page 25-2 and “Template Makefiles and Make Options” on page 18-32 .

Setting Up Rapid Simulation Input Data

In this section...

“Introduction” on page 42-12

“Creating a MAT-File That Includes a Model Parameter Structure” on page 42-13

“Creating a MAT-File for a From File Block” on page 42-17

“Creating a MAT-File for an Import Block” on page 42-18

Introduction

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 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.

Provided by an Import block

Create a MAT-file that adheres to one of the three data file formats that the Import 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 Import 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 Import block data must be present if a From File block or Import block exists in your model.

Creating 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.

Getting the Parameter Structure for a Model

Get the global parameter structure (*model_P*) for a model by calling the function `rsimgetrtp`.

```
param_struct = rsimgetrtp('model')
```

Argument	Description
<i>model</i>	The model for which you are running the rapid simulations.

The `rsimgetrtp` function forces an update diagram action for the specified model and returns a structure that contains the following fields.

Field	Description
<code>modelChecksum</code>	A four-element vector that encodes the structure of the model. The Real-Time Workshop software uses the checksum to check whether the structure of the model has changed since the RSim executable was generated. If you delete or add a block, and then generate a new <code>model_P</code> vector, the new checksum no longer matches the original checksum. 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.
<code>parameters</code>	A structure that contains the model's global parameters.

The parameter structure contains the following information.

Field	Description
<code>dataTypeName</code>	The name of the parameter data type, for example, <code>double</code>
<code>dataTypeID</code>	An internal data type identifier used by the Real-Time Workshop software
<code>complex</code>	The value 0 if real; 1 if complex
<code>dtTransIdx</code>	Internal data index used by Real-Time Workshop
<code>values</code>	A vector of the parameter values associated with this structure
<code>map</code>	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 <code>rsimsetrtpparam</code> , 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 Real-Time Workshop 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 demo 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]
```

Saving 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 “Running Rapid Simulations” on page 42-24.

Converting 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 any 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 “Changing Block Parameters for an RSim Simulation” on page 42-34.

Creating 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.

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.

Creating a MAT-File for an Import Block

You can use a MAT-file as the input data source for an Import block.

The format of the data in the MAT-file must adhere to one of the three column-based formats listed in the following table. The table lists the formats in order from least flexible to most flexible.

Format	Description
Single time/data matrix	<ul style="list-style-type: none">• Least flexible.• One variable.• Two or more <i>columns</i>. Number of columns must equal the sum of the dimensions of all root Import blocks plus 1. First column must contain monotonically increasing time points. Other columns contain data points that correspond to the time point in a given row.• Data of type <code>double</code>. <p>For an example, see Single time/data matrix in the following procedure, step 4. For more information, see “Importing Data Arrays” in the Simulink documentation.</p>

Format	Description
Signal-and-time structure	<ul style="list-style-type: none"> More flexible than the single time/data matrix format. One variable. Must contain two top-level fields: <code>time</code> and <code>signals</code>. The <code>time</code> field contains a <i>column</i> vector of the simulation times. The <code>signals</code> field contains an array of substructures, each of which corresponds to an Import block. The substructure index corresponds to the Import block number. Each <code>signals</code> substructure must contain a field named <code>values</code>. The <code>values</code> field must contain an array of inputs for the corresponding Import block, where each input corresponds to a time point specified by the <code>time</code> field. If the <code>time</code> field is set to an empty value, clear the check box for the Import block Interpolate data parameter. No data type limitations, but must match Import block settings. <p>For an example, see Signal-and-time structure in the following procedure, step 4. For more information on this format, see “Importing Data Structures” in the Simulink documentation.</p>

Format	Description
Per-port structure	<ul style="list-style-type: none"> Most flexible Multiple variables. Number of variables must equal the number of Import blocks. Consists of a separate structure-with-time or structure-without-time for each Import block. Each Import block data structure has only one <code>signals</code> field. To use this format, in the Input text field, enter the names of the structures as a comma-separated list, <code>in1</code>, <code>in2</code>,..., <code>inN</code>, where <code>in1</code> is the data for your model’s first port, <code>in2</code> for the second port, and so on. Each variable can have a different time vector. If the <code>time</code> field is set to an empty value, clear the check box for the Import block Interpolate data parameter. No data type limitations, but must match Import block settings.

Format	Description
	<ul style="list-style-type: none"> To save multiple variables to the same data file, you must save them in the order expected by the model, using the -append option. <p>For an example, see Per-port structure in the following procedure, step 4. For more information, see “Importing Data Structures” in the Simulink documentation.</p>

The supported formats and the following procedure are illustrated in `rtwdemo_rsim_i`.

To create a MAT-file for an Import block:

- 1 Choose one of the preceding data file formats.
- 2 Update Import block parameter settings and specifications to match specifications of the data to be supplied by the MAT-file.

By default, the Import 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 Import block in the Simulink documentation.

- 3 Build an RSim executable for the model. The Real-Time Workshop 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 ensure correct ordering of 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.

Programming 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 any 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 -s 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run2.mat -o results2.mat -s 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run3.mat -o results3.mat -s 10.0
rtwdemo_rsimtf -f rtwdemo_rsimtf.mat=run4.mat -o results4.mat -s 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 “Running Rapid Simulations” on page 42-24. For an example of a rapid simulation batch script, see the demo `rtwdemo_rsim_batch_script`.

Running Rapid Simulations

In this section...

- “Introduction” on page 42-24
- “Requirements for Running Rapid Simulations” on page 42-26
- “Setting a Clock Time Limit for a Rapid Simulation” on page 42-26
- “Overriding a Model Simulation Stop Time” on page 42-27
- “Reading the Parameter Vector into a Rapid Simulation” on page 42-28
- “Specifying New Signal Data File for a From File Block” on page 42-28
- “Specifying Signal Data File for an Inport Block” on page 42-31
- “Changing Block Parameters for an RSim Simulation” on page 42-34
- “Specifying a New Output File Name for a Simulation” on page 42-36
- “Specifying New Output File Names for To File Blocks” on page 42-36

Introduction

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.

Operating System Command Line

```
rtdemo_rsimtf
```

MATLAB Command Line

```
!rtdemo_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	<code>model -f oldfilename.mat=newfilename.mat</code>
Print a summary of the options for RSim executable targets	<code>executable filename -h</code>
Read input data for an Import block from a MAT-file	<code>model -i filename.mat</code>
Time out after n clock time seconds, where n is a positive integer value	<code>model -L n</code>
Write MAT-file logging data to file <code>filename.mat</code>	<code>model -o filename.mat</code>
Read a parameter vector from file <code>filename.mat</code>	<code>model -p filename.mat</code>
Override the default TCP port (17725) for external mode	<code>model -port TCPport</code>
Write MAT-file logging data to a MAT-file other than the MAT-file used for the previous simulation	<code>model -t oldfilename.mat=newfilename.mat</code>
Run the simulation until the time value <code>stoptime</code> is reached	<code>model -tf stoptime</code>
Run in verbose mode	<code>model -v</code>
Wait for the Simulink engine to start the model in external mode	<code>model -w</code>

The following sections use the `rtwdemo_rsimtf` demo 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 demo.

- 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',...
    'rsimdemos','rsim_tfdata.mat'),pwd);
```

Requirements for Running Rapid Simulations

- You can run the RSim executable on any computer configured to run MATLAB and for which the MATLAB and Simulink installation directory is accessible to the `RSim.exe`. 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 Sun Solaris™ platforms, to run an RSim executable generated for a model that uses variable-step solvers in a separate shell, define the `LD_LIBRARY_PATH` environment variable to provide the path to the MATLAB installation directory, as follows:

```
% setenv LD_LIBRARY_PATH /apps/matlab/bin/sol64:$LD_LIBRARY_PATH
```

- On GNU Linux platforms, to run an RSim executable, define the `LD_LIBRARY_PATH` environment variable to provide the path to the MATLAB installation directory, 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 directories `bin/mac` and `sys/os/mac` under the MATLAB installation directory. 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`.

Setting 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 is terminate. You see one of the following messages:

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 Real-Time Workshop configuration to use this option.

Overriding 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.

Reading 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 “Creating a MAT-File That Includes a Model Parameter Structure” on page 42-13. You can then specify the MAT-file in the command line with the -p option.

For example:

- 1 Build an RSim executable for the demo `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)
```

Specifying 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 “Creating a MAT-File for a From File Block” on page 42-17.

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 demo `rtwdemo_rsimtf`.

- 3** Run the executable.

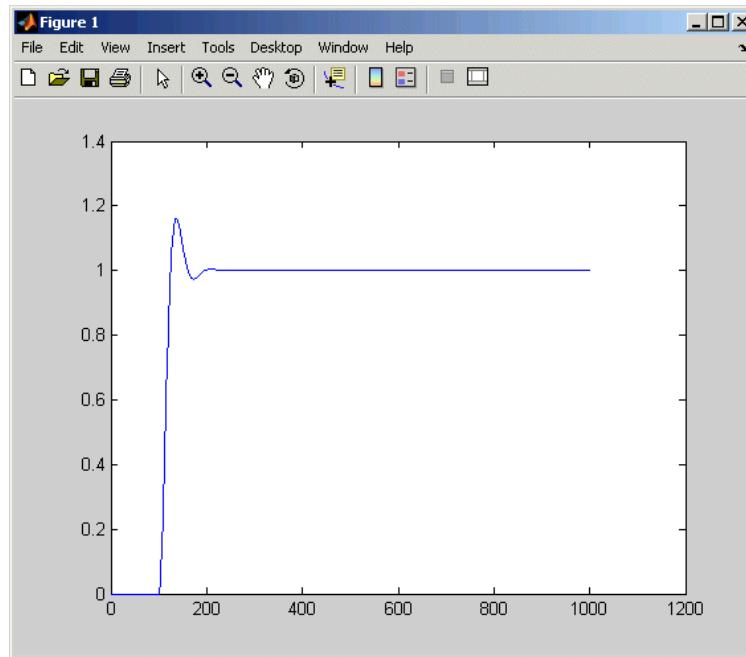
```
!rtwdemo_rsimtf
```

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;
```

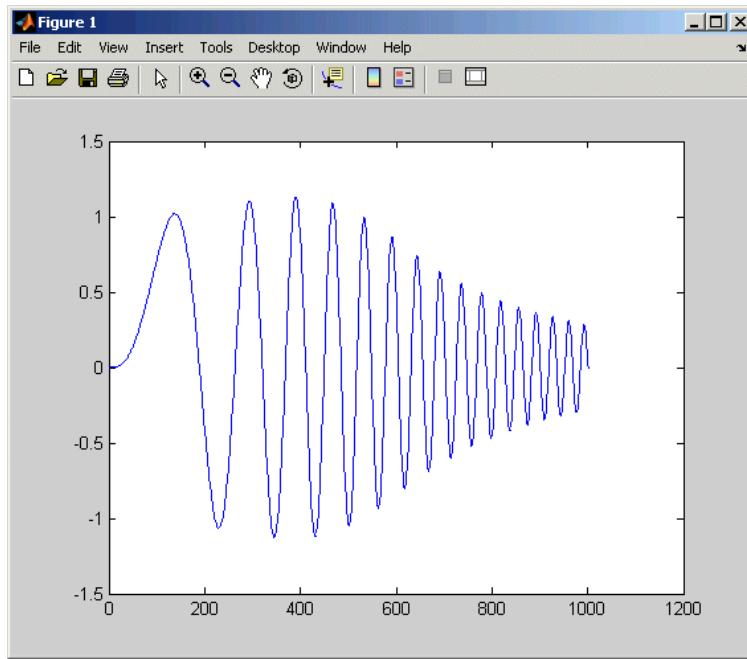
6 Run a rapid simulation with the new data by using the `-f` option to replace the original file, `rsim_tfdata.mat`, with `newfrom.mat`.

```
!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 Import block (see

“Creating a MAT-File for an Import Block” on page 42-18) 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
```

Specifying Signal Data File for an Import Block

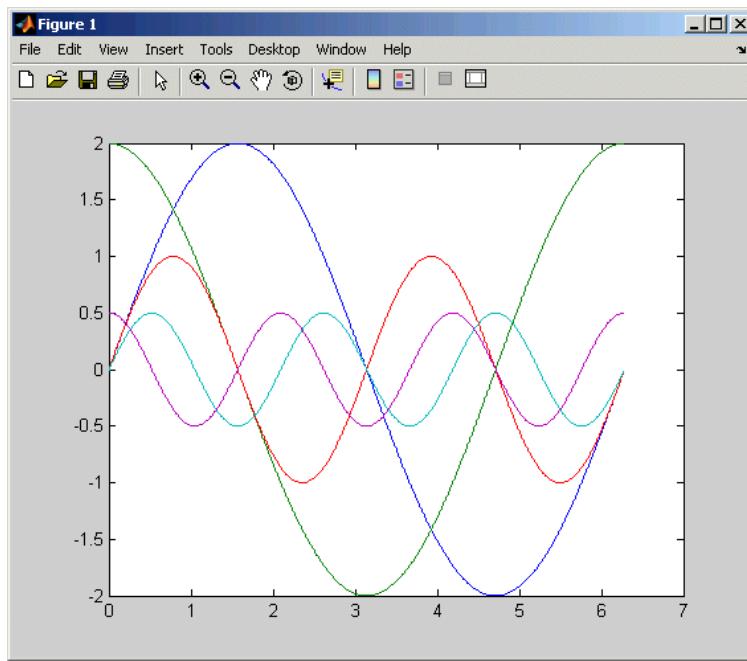
If your model’s input data source is an Import block, you can feed the block with input data during simulation from a single MAT-file or you can change the MAT-file from one simulation to the next. Each MAT-file must adhere to one of the three formats described in “Creating a MAT-File for an Import Block” on page 42-18.

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 `rtdemo_rsim_i`.
- 2** Check the Import block parameter settings. The following Import block data parameter settings and specifications that you specify for the workspace variables must match settings in the MAT-file, as indicated in “Configuring Import Blocks to Provide Rapid Simulation Source Data” on page 42-8:
 - **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 “Creating a MAT-File for an Inport Block” on page 42-18. 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.

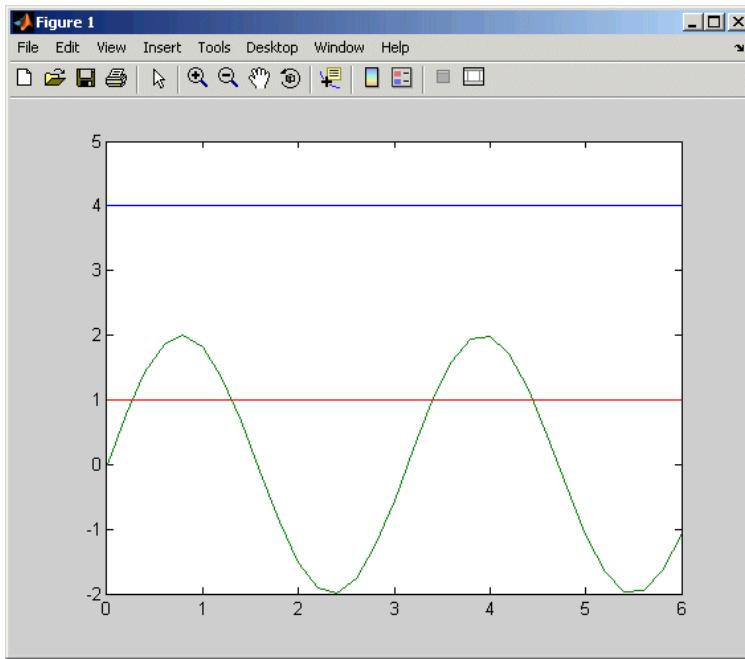
```
!rtwdemo_rsim_i -i rsim_i_single_struct.mat

** Starting model 'rtwdemo_rsim_i' @ Tue Dec 27 14:01:20 2005
*** rsim_i_single_struct.mat is successfully 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);
```



Changing Block Parameters for an RSim Simulation

As described in “Creating a MAT-File That Includes a Model Parameter Structure” on page 42-13, 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 any model parameter, including parameters that include *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 Real-Time Workshop 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 no longer visible

in the generated code. 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 “Creating a MAT-File That Includes a Model Parameter Structure” on page 42-13.

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 “Converting the Parameter Structure for Running Simulations on Varying Data Sets” on page 42-15, 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
```

Specifying a New Output File Name for a Simulation

If you have specified any 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 demo *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 any From File block data, is run.

Specifying 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 MATLAB Fcn 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 all 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.

Verifying Generated Code

- “About Generated Code Verification” on page 43-2
- “Tracing Generated Code” on page 43-3
- “Profiling Code Performance” on page 43-7

About Generated Code Verification

You can verify generated code with at least three different goals in mind:

- Does the code generated for specific areas of a model appear as expected? See “Tracing Generated Code” on page 43-3.
- Does the generated code produce any run-time errors? Are the run-time results correct? See “Checking Code Correctness”.
- Does the generated code meet performance requirements? See “Profiling Code Performance” on page 43-7.

Tracing Generated Code

In this section...

- “About Code Traceability” on page 43-3
- “Format of Traceability Tags” on page 43-3
- “Examples of Tagged Code” on page 43-4
- “Tracing Code To Blocks Using `hilite_system`” on page 43-4
- “Traceability Limitations” on page 43-5

About Code Traceability

The Real-Time Workshop code generator produces the HTML code generation report for a model. The report includes the following resources that support code traceability:

- Code element hyperlinks (indicated with underlining) that you can use to trace through and toggle between generated source and header files.
- Tags in code comments that identify blocks from which lines of code are generated.

Format of Traceability Tags

A traceability tag appears in a comment above the corresponding line of generated code. The format of the tags is `<system>/block_name`, where

- *system* is one of the following:
 - The string `Root`
 - A unique system number assigned by the Simulink engine
- *block_name* is the name of the source block

The Real-Time Workshop code generator documents all the tags for a model in the comments section of the generated header file `model.h`. For example, the following comment appears in the header file for model, `foo`, that has a subsystem `Outer` and a nested subsystem `Inner`:

```
/* Here is the system hierarchy for this model.
```

```
*  
* <Root> : foo  
* <S1>   : foo/Outer  
* <S2>   : foo/Outer/Inner  
*/
```

Examples of Tagged Code

The following code shows a tag comment adjacent to a line of code. This code is generated from a Gain block at the root level of a source model:

```
/* Gain: '<Root>/UnDeadGain1' */  
rtb_UnDeadGain1_h = dead_gain_U.In1 *  
dead_gain_P.UnDeadGain1_Gain;
```

The following code shows a tag comment adjacent to a line of code. This code is generated from a Gain block within a subsystem one level below the root level of the source model:

```
/* Gain Block: <S1>/Gain */  
dead_gain_B.temp0 *= (dead_gain_P.s1_Gain_Gain);
```

Tracing Code To Blocks Using `hilite_system`

To trace generated code back to the corresponding source system or block in a model:

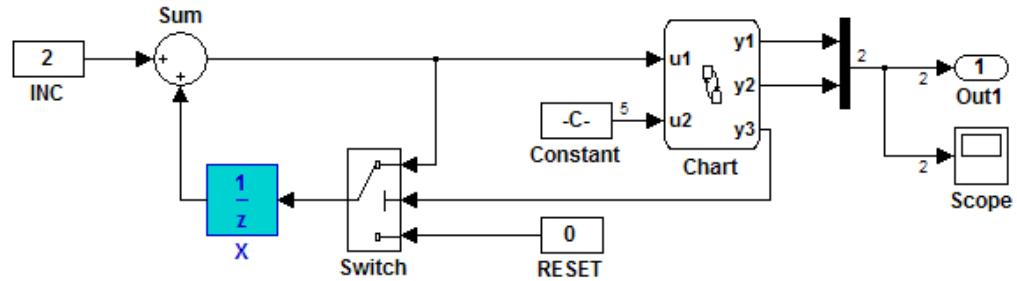
- 1** Open the source model.
- 2** In an editor or in the context of an HTML code generation report, open a generated source or header file.
- 3** As you review lines of code, note traceability tags that correspond to code of interest. In the model diagram, highlight the block that corresponds to a line of code by entering the `hilite_system` command with the traceability tag that marks that code. The command changes the color of the outline and label of the corresponding block and fills the block with shading.

For example, suppose the traceability tag `<Root>/x` appears in code generated for model `foo`. In this case, enter the following commands to highlight block `x`:

```
hilite_system('foo/x')
```

```
hilite_system('<Root>/x')
```

The following figure shows block x highlighted.



You can also use the `hilite_system` command to highlight a subsystem. Specify the subsystem using its system name or label. For example, you can highlight subsystem S2, which is labeled bar, using either of the following commands:

```
hilite_system('<S2>')
```

```
hilite_system('<Root>/bar')
```

If you have a Real-Time Workshop Embedded Coder license, HTML code generation reports display hyperlinks in “Regarding,” “Outport,” and other comment lines. In addition, you can trace subsystems and blocks to corresponding generated code. For more information, see “About Traceability” in the Real-Time Workshop Embedded Coder documentation.

For more information on using HTML code generation reports for traceability analysis, see “Reviewing Generated Code” on page 23-3.

Traceability Limitations

In HTML code generation reports, the traceability report sections have the following limitations:

- Under the following conditions, code-to-model traceability is disabled for a block if the block name contains:
 - A single quote (').

- An asterisk (*), that causes a name-mangling ambiguity relative to other names in the model. This name-mangling ambiguity occurs if in a block name or at the end of a block name, an asterisk precedes or follows a slash (/).
- The character (char(255)).
- After building a model, if you move the .mdl file to a different directory, traceability might not work.

Profiling Code Performance

In this section...

[“About Profiling Code Performance” on page 43-7](#)

[“How to Profile Code Performance” on page 43-7](#)

[“Profiling Limitation” on page 43-9](#)

About Profiling Code Performance

By profiling the performance of generated code, you can better assure 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 correct 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.

How to Profile Code Performance

You can profile code generated with Real-Time Workshop technology by using a Target Language Compiler (TLC) hook function interface.

To use the profile hook function interface:

- 1 For your target, create a TLC file that defines the following hook functions. Write the functions so that they specify profiling code. The code generator adds the hook function code to code generated for atomic systems in the model.

Function	Input Arguments	Output Type	Description
ProfilerHeaders	void	Array of header file names	Return an array of the header file names to be included in the generated code.
ProfilerTypedefs	void	typedefs	Generate code statements for profiler type definitions.

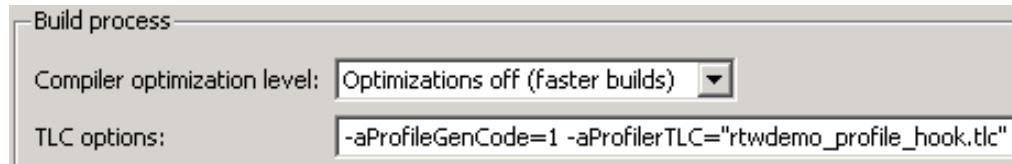
Function	Input Arguments	Output Type	Description
ProfilerGlobalData	system	Global data for the specified system	Generate code statements that declare global data.
ProfilerExternDataDecls	system	extern declarations for the specified system	Generate code statements that create global extern declarations.
ProfilerSysDecl	system, functionType	Declarations for the specified system for the specified functionType	Generate code for variable declarations needed within the scope of an atomic subsystem Output, Update, OutputUpdate, or Derivatives function.
ProfilerSysStart	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.
ProfilerSysEnd	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.
ProfilerSysTerminate	system	Profiler termination code for the specified system	Generate code that terminates profiling (and possibly reports results) for an atomic subsystem.

For an example, open the .tlc file
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
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 in the Configuration Parameters dialog box, in **Real-Time Workshop > TLC options**.



- 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 appropriate 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, run the demo `rtwdemo_profile`. For details on programming a `.tlc` file and defining TLC configuration variables, see the Target Language Compiler documentation.

Profiling Limitation

The TLC hook function interface for profiling code performance does not support the S-function target (`rtwsfcn.tlc`).

Testing and Refining a Model With Rapid Prototyping

- “About Rapid Prototyping” on page 44-2
- “Rapid Prototyping System Simulations” on page 44-4
- “Rapid Prototyping a Real-Time System” on page 44-7

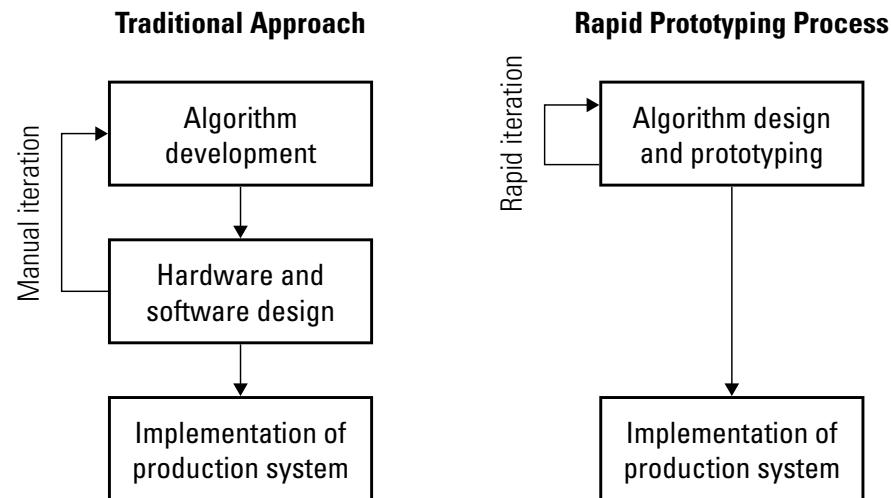
About Rapid Prototyping

Rapid prototyping is an application development technique for testing new ideas and research. During rapid prototyping, you generate, run, and tune code for a model.

You rapid prototype a refined concept model on inexpensive PC or nontarget hardware, including off-the-shelf I/O cards. You might deploy code for a component model (algorithm or controller) on a real-time simulator connected to system hardware (the physical plant or vehicle being controlled).

While rapid prototyping, you place less emphasis on code efficiency and I/O latency. Primary goals are to:

- Combine algorithm, software, and hardware design phases of system design, eliminating potential bottlenecks
- Refine model designs by rapidly iterating between algorithm design and prototyping
- Validate whether a component can adequately control a physical system in non-real time
- Evaluate system performance before laying out hardware, coding production software, or committing to a fixed design
- Reduce development costs by rapidly iterating solutions before building expensive hardware



Rapid Prototyping System Simulations

In this section...

“About System Simulation” on page 44-4

“Goals of System Simulation” on page 44-4

“Running System Simulations” on page 44-5

About System Simulation

System simulation involves integrating the generated code for one or more component models into a system or environment model that represents the context in which the components eventually run.

When you rapid prototype a system simulation, you simulate the model in nonrealtime — that is, Simulink uses a computed time vector to step the model. After computing the outputs for a given time value, Simulink immediately repeats the computations for the next time value. Simulink repeats this process until the simulation reaches the stop time.

For system simulation, it might be necessary to share components with a different group or integrate them into another execution environment for additional analysis and model iterations.

Goals of System Simulation

Assuming that you have documented functional requirements, refined concept models, and verified that the environment or system into which components are to be added exists, you can use system simulation to:

- Refine component models
- Refine the system model
- Validate functionality of component models in non-real time
- Test concept models

Running System Simulations

To integrate a component model into another environment for system simulation:

- 1 Generate source code from the component model. Consider the following design issues as you configure the model for code generation.

Engineering Tasks	Related Product Information	Demos
Schedule the generated code	Chapter 5, “Scheduling Considerations” “Handling Asynchronous Events” on page 5-34	<code>rtwdemos</code> , select Multirate Support folder
Specify function boundaries	“Creating Subsystems” on page 3-37	<code>rtwdemo_atomic</code> <code>rtwdemo_ssreuse</code> <code>rtwdemo_filepart</code>
Specify components and boundaries for design and incremental code generation	Chapter 3, “Architecture Considerations”	<code>rtwdemo_mdlref_top</code>

2 Supply the code, dependent files, and data for building in another environment. One approach is to use the `packNGo` function to create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer. For an example, see `rtwdemo_buildinfo`. For more information, see “Relocating Code to Another Development Environment” on page 25-29.

3 Simulate and test the system.

4 Refine the component model and iterate until the component model produces expected results in the context of the system.

At some point in this process, you might add hardware.

If you have a Real-Time Workshop Embedded Coder license, an alternative approach is to generate a shared library for a component model and supply the

library to others for dynamic linking into a system. For more information, see Chapter 7, “Protecting Intellectual Property in Shared Model Components”.

Rapid Prototyping a Real-Time System

In this section...

[“About Real-Time Rapid Prototyping” on page 44-7](#)

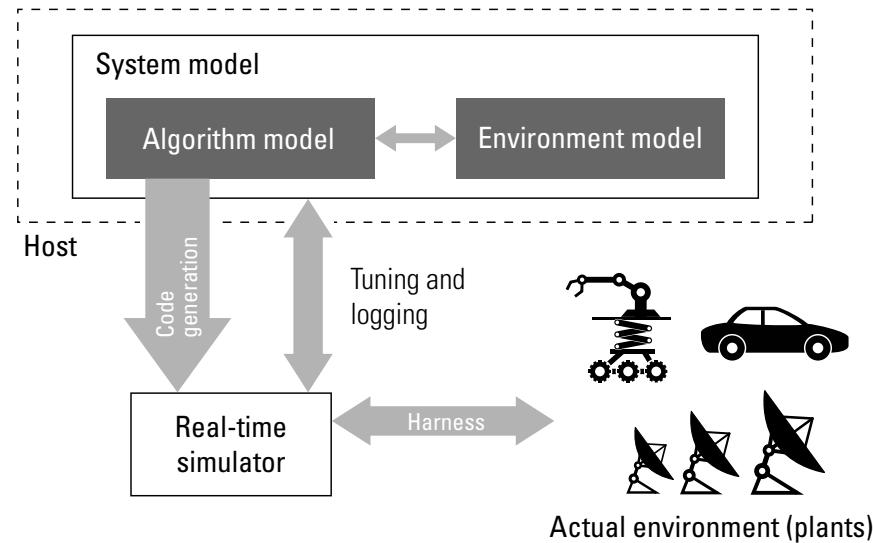
[“Goals of Real-Time Rapid Prototyping” on page 44-8](#)

[“Refining Component Code With Real-Time Rapid Prototyping” on page 44-8](#)

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 xPC Target 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

Refining Component 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 xPC Target 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 xPC Target product or other Real-Time Workshop technology that you can integrate into a development environment.
- 2 Use Real-Time Workshop system target files to generate code that you can deploy onto a real-time simulator. See the following information.

Engineering Tasks	Related Product Information	Demos
Generate code for real-time rapid prototyping	<p>Chapter 18, “Selecting and Configuring a Target” in the Real-Time Workshop documentation</p> <p>Real-Time Workshop Embedded Coder</p> <p>“Developing Models for Code Generation” in the Real-Time Workshop Embedded Coder documentation</p> <p>Chapter 34, “Interfacing With a Real-Time Operating System” in the Real-Time Workshop documentation</p>	<code>rtwdemo_counter</code> <code>rtwdemo_async</code>
Generate code for rapid prototyping in hard real time, using PCs	xPC Target xPC Target documentation	<code>help xpcdemos</code>
Generate code for rapid prototyping in soft real time, using PCs	Real-Time Windows Target Real-Time Windows Target documentation	<code>rtvdp</code> (and others)

3 Monitor signals, tune parameters, and log data.

Verifying Generated Source Code With Software-In-the-Loop Simulation

- “About Software-In-the-Loop Simulation” on page 45-2
- “Maintaining Bit-True Agreement Between Host Simulations and Code for Target Deployment” on page 45-3
- “Setting Up and Running SIL Simulations Using a Hand-Written S-Function Wrapper” on page 45-5

About Software-In-the-Loop Simulation

Software-in-the-loop (SIL) simulation involves simulating target behavior for a model on a MATLAB host system. With the Real-Time Workshop product, you can run such simulations using S-function wrappers. In addition, if processor word sizes differ between host and target platforms (for example, a 32-bit host and a 16-bit target), you can configure the model to use hardware emulation to simulate target behavior on the host platform.

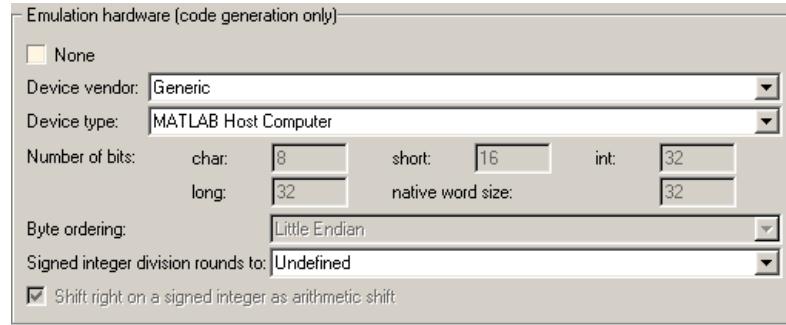
Hardware emulation guarantees bit-true agreement for integer and fixed-point operations between the results of simulations run on the MATLAB host system and the results of generated code executed on the target deployment system. Generated code emulates the target behavior. The code that you generate for simulation might contain additional code, such as data type casts, to ensure behavior that is consistent with the target environment.

Maintaining Bit-True Agreement Between Host Simulations and Code for Target Deployment

To guarantee bit-true agreement for integer and fixed-point operations between the results of simulations that run on the MATLAB host system and the results of generated code executed on the target deployment system, configure your model for hardware emulation.

To configure a model for SIL simulation that uses hardware emulation, set the following model configuration parameters.

Set...	To...
Hardware Implementation > Emulation hardware > None	Cleared
Hardware Implementation > Emulation hardware > Device vendor	Generic
Hardware Implementation > Emulation hardware > Device type	MATLAB Host Computer
Real-Time Workshop > SIL and PIL Verification > Enable portable word sizes (Real-Time Workshop Embedded Coder only)	Cleared



You might also want to adjust parameters on the Optimization pane, in the **Code Generation** section.

Note After completing your SIL simulation on the MATLAB host system, and before target deployment, you must select **None** for **Hardware Implementation > Emulation hardware** and then regenerate code for the target.

Setting Up and Running SIL Simulations Using a Hand-Written S-Function Wrapper

To set up and run SIL tests, using a hand-written S-function wrapper and the Real-Time Workshop product:

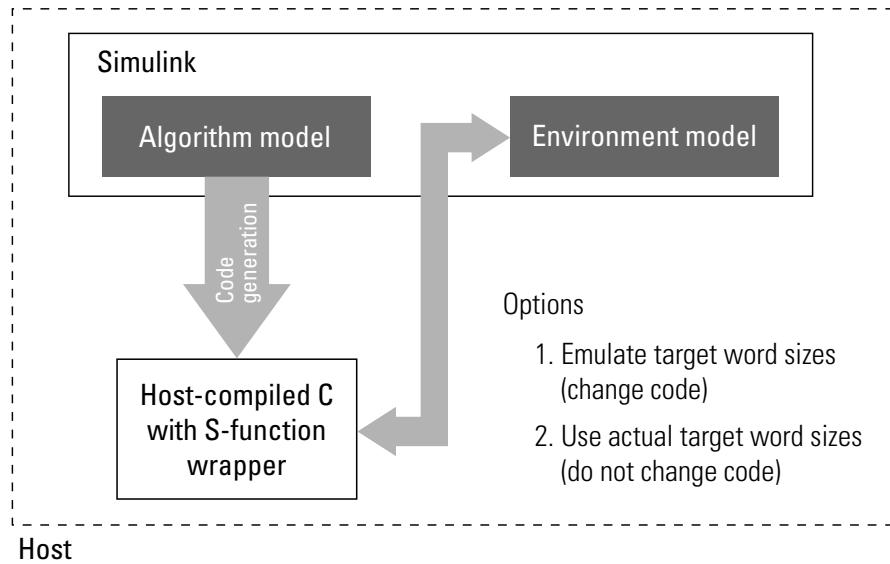
- 1** If you need to guarantee bit-true agreement for integer and fixed-point operations between the results of host system simulations and the results of generated code executed on the target deployment system, set the model configuration parameters as specified in “Maintaining Bit-True Agreement Between Host Simulations and Code for Target Deployment” on page 45-3.
- 2** Generate code for the component model.
- 3** Write an S-function wrapper in C; see “Writing Wrapper S-Functions” on page 32-12.

The S-function wrapper code serves as an interface that calls your component C code.

- 4** Use the Simulink `mex` command to create a block to represent the S-function. For example:

```
mex sfcn_wrapper.c component_code.c
```

Simulink generates an S-function block.



- 5** Add the generated S-function block to an environment or test harness model that supplies test vectors or stimulus input.
- 6** Run SIL tests on the host system by simulating the environment model.
- 7** Verify that the generated code captured in the S-function block provides the same result as the original subsystem.

If you have a Real-Time Workshop Embedded Coder license, you can carry out a SIL simulation by using either the SIL simulation mode or a SIL block. See “Verifying Generated Source Code With Software-in-the-Loop Simulation”.

Verifying a System With Hardware-In-the-Loop Simulation

- “About Hardware-In-the-Loop Simulation” on page 46-2
- “Setting Up and Running HIL Simulations” on page 46-4

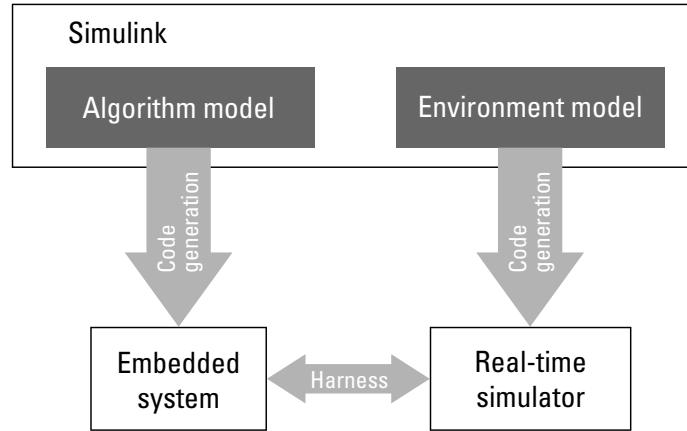
About Hardware-In-the-Loop Simulation

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 Real-Time Workshop 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 The MathWorks is the xPC Target product. Several third-party products are also available for use as HIL platforms. The xPC Target product offers hard real-time performance for any PC with Intel® or AMD® 32-bit processors functioning as your real-time target. The

xPC Target product enables you to add I/O interface blocks to your models and automatically generate code with Real-Time Workshop technology. The xPC Target product can download the code to a second PC running the xPC Target real-time kernel. System integrator solutions that are based on xPC Target are also available. For more information about xPC Target, see the xPC Target documentation.

Setting Up and Running 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:
 - Chapter 18, “Selecting and Configuring a Target ”
 - Chapter 34, “Interfacing With a Real-Time Operating System ”
- 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 successfully simulate the system after including all components that require testing.

A

Troubleshooting

Troubleshooting Compiler Configurations

In this section...

- “Compiler Version Mismatch Errors” on page A-2
- “Generated Executable Image Produces Incorrect Results” on page A-2
- “Compile-Time Errors” on page A-3

Compiler Version Mismatch Errors

Explanation

You received a version mismatch error when you compiled code generated by the Real-Time Workshop 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 “Choosing and Configuring a Compiler” on page 25-2.
- 3 Rebuild the model.

Generated Executable Image Produces Incorrect Results

Explanation

You applied compiler optimizations when you used Real-Time Workshop to generate an executable image. However, due to a compiler defect, the optimizations caused the executable image to produce incorrect results, even though the generated code is correct.

User Action

Do one of the following:

- Lower the compiler optimization level.
 - 1 Select **Custom** for the Model Configuration parameter **Real-Time Workshop > 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 **Real-Time Workshop > Compiler optimization level**.
 - 2 Rebuild the model.

For more information, see “Controlling Compiler Optimization Level and Specifying Custom Optimization Settings” on page 25-10 and your compiler documentation.

Compile-Time Errors

Explanations

- You received a compiler configuration error.
- Environment variables for your make utility, compiler, or linker are not set up correctly. 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 **Real-Time Workshop > 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 “Choosing and Configuring a Compiler” on page 25-2).
- 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 correct target.

Examples

Use this list to find examples in the documentation.

Model Reference

“Code Reuse and Model Blocks with Root Import or Outport Blocks” on page 3-74

Models

“Example: Single-Tasking and Multitasking Execution of a Model” on page 5-27

“Dual-Model Approach: Simulation” on page 5-40

“Spawning a Wind River Systems VxWorks Task” on page 5-45

“Block Execution Order” on page 6-7

“Controlling Signal Object Code Generation By Using Typed Commands” on page 12-13

“Other Optimization Tools and Techniques” on page 26-7

“Inlining Invariant Signals” on page 26-41

“Configuring a Loop Unrolling Threshold” on page 26-45

Timing Services

“Elapsed Timer Code Generation Example” on page 5-77

Data Management

“Nontunable Parameter Storage” on page 10-3

“Tunable Expressions in Masked Subsystems” on page 10-16

“Signals with Auto Storage Class” on page 11-6

“Symbolic Naming Conventions for Signals in Generated Code” on page 11-14

Model Code Packaging

“Code Packaging Example” on page 25-32

Optimizations

“Expression Folding Example” on page 26-11
“Multiple Tables with Common Inputs” on page 26-26
“Optimizing Data Type Usage” on page 26-33
“Expression Folding for Blocks with Multiple Outputs” on page 32-70

Advanced Code Generation

“Example Build Process Customization Using sl_customization.m” on page 27-32

Custom Code

“Example: Using a Custom Code Block” on page 31-6

S-Functions

“TLC S-Function Wrapper” on page 32-17
“Writing Fully Inlined S-Functions” on page 32-23
“Multiport S-Function Example” on page 32-23
“S-Function RTWdata” on page 32-31
“The Direct-Index Lookup Table Example” on page 32-33

Interfaces

“Generating Example C API Files” on page 35-9

“Using the C API in an Application” on page 35-21

Index

A

absolute time computation 5-70
`addLibs` field 32-101
algorithm models
 integrating for real-time rapid prototyping 44-7
 integrating for system simulation 44-4
APIs
 timer services 5-72
application modules
 application-specific components 3-33
 definition of 3-22
 system-independent components 3-27
ASAP2 files
 customizing 36-12
 data attributes required for 36-4
 generating 36-7
 structure of 36-19
 targets supporting 36-3
asynchronous tasks
 timers for 5-71
atomic subsystem 3-37
automatic S-function generation 3-104
 See also S-function target

B

block states
 Simulink data objects and 14-11
 State Properties dialog box and 14-6
 storage and interfacing 14-1
 storage classes for 14-4
 symbolic names for 14-8
block-based code integration
 with S-functions 32-90
blocks
 Custom Code 31-2
 depending on absolute time 5-81
 Model Header 31-3
 Model Source 31-3

Rate Transition 5-13
blocks, Simulink
 support for 4-2
Browse button
 on Real-Time Workshop pane 18-29
buffer reuse option 26-40
build directory
 contents of 25-21
 naming convention 25-20
build directory optional contents
 C API files 25-21
 HTML report files 25-21
 model.rtw 25-21
 object files 25-21
 subsystem code modules 25-21
 TLC profiler files 25-21
build process
 COM automation of 21-138
 controlling 27-7
 files and directories created 25-15
 interfacing to development tools
 integrated development environments 21-131
 make utilities 21-131
 passing information in 21-33
 phases of 21-30
 steps in 25-25
build specification 32-100

C

C API
 files used in 35-5
 for S-functions 5-73
 generating files 35-3
 introduction 35-1
 mapping to real-time model 35-19
 using for your application 35-21
C language
 selecting 19-1

C++ encapsulation interface control, custom
 target support for 21-128

C++ language
 selecting 19-1

checksums
 and S-Function target 3-109
 for models 3-109
 subsystem 3-109

code
 integrating existing 29-2
 build support for 32-90

code files
 porting 25-29
 relocating 25-29

code format
 choosing 18-10
 embedded 18-18
 real-time 18-14
 real-time malloc 18-16
 S-function 18-17

code generation
 from nonvirtual subsystems 3-37
 TLC variables for 21-42

code generation options
 Application lifespan (days) 26-51
 Boolean logic signals 26-39
 buffer reuse 26-40
 See also signal storage reuse 26-40
 Compiler optimization level 25-10
 create code generation report 23-2
 Custom compiler optimization flags 25-10
 expression folding 26-10
 Generate makefile option 25-11
 GRT compatible call interface 18-23
 inline invariant signals 26-41
 inline parameters 26-43
 local block outputs 26-40
 See also signal storage reuse 26-40
 loop rolling threshold 26-45
 MAT-file variable name modifier 33-17

retain .rtw file 17-1
 show eliminated blocks 20-2
 signal storage reuse 26-39
 See also local block outputs 26-39

Solver pane 5-83

TLC options 25-11

Use memcpy for vector assignment 26-47

verbose builds 17-1

code reuse
 diagnostics for 3-81
 enabling 3-73

code tracing
 by using hilite_system command 43-4
 by using HTML reports 43-4

code, generated
 testing in software environment 45-2
 testing in system environment 46-2

combining models
 by using grt_malloc target 3-114
 in Real-Time Workshop Embedded Coder
 target 3-114

communication
 external mode 37-3

compilation 27-2
 customizing 27-3

compilation report keyboard shortcuts
 emlc 28-85

compiler
 configuring 25-2

Compiler optimization level control, custom
 target support for 21-122

compiler options
 specifying 27-3

compilers
 list of supported 25-2
 MEX 25-2
 optimization settings A-2
 supported for generating C code with
 emlc 28-9

- supported for generating MEX functions
 - with emlc 28-9
- version mismatches for A-2
- component models
 - integrating for real-time rapid prototyping 44-7
 - integrating for system simulation 44-4
- configuration objects
 - for emlc 28-16
- configuration parameters
 - TargetLibSuffix**
 - controlling suffix applied to library names with 27-10
 - TargetPreCompLibLocation**
 - controlling location of precompiled libraries with 27-8
- Configuration Parameters dialog box
 - Real-Time Workshop pane 16-2
 - specifying nonvirtual code generation with 3-37
- Solver options pane 5-83
- configuring your environment for code generation
 - emlc configuration objects 28-16
- continuous states, integration of 3-26
- controller models
 - integrating for real-time rapid prototyping 44-7
 - integrating for system simulation 44-4
- controlling runtime checks
 - Embedded MATLAB Coder 28-101
- counters
 - in triggered subsystems 5-72
 - time 5-70
- Create code generation report 23-2
- cross-development
 - relocating files for 25-29
- custcode** command 31-2
- custom code
 - build support for 32-90
 - integrating with C MEX S-functions 32-25
 - integrating with generated code 29-2
- Custom Code blocks 31-2
 - example 31-6
 - in subsystems 31-9
- Custom Code library
 - overview 31-2
- custom target
 - components of 21-12
 - application 21-13
 - code 21-13
 - control files 21-15
 - device drivers 21-15
 - interrupt service routines 21-14
 - main program 21-14
 - run-time interface 21-13
 - purpose of 21-2
- custom target configuration
 - tutorial 21-71

D

- data representation and storage
 - deciding on 9-1
- Data Store Memory blocks
 - Simulink data objects and 15-6
- data structures in generated code
 - block I/O 3-20
 - block parameters 3-20
 - block states 3-20
 - external inputs 3-20
 - external outputs 3-20
- debugging runtime errors
 - Embedded MATLAB 28-104
- declaration code 31-5
- design specification
 - developing 2-1
- development environments
 - supporting multiple 21-62
- direct-index lookup table
 - algorithm 32-32

example 32-33
 directories
 used in build process 25-2
 directory
 precompiled library 32-100
 discrete states
 initializing 12-18
 displaying target options 21-69
 domain
 installing products for 2-1
`dt_info.h` 22-11

E

elapsed time computation 5-70
 elapsed time counters 5-70
 in triggered subsystems 5-72
 elapsed timer
 example 5-77
 Embedded MATLAB
 debugging runtime errors 28-104
 Embedded MATLAB blocks
 and Stateflow optimizations 26-46
 Embedded MATLAB Coder
 combining property specifications 28-37
 controlling runtime checks 28-101
 how to disable runtime checks 28-102
 specifying general properties of primary
 inputs 28-37
 when to disable runtime checks 28-102
`emlc`
 compilation report keyboard shortcuts 28-85
 supported compilers for generating C
 code 28-9
 supported compilers for generating MEX
 functions 28-9
 Euler integration algorithm 3-26
 examples
 direct-index lookup table 32-33
 multiport S-function 32-23

execution code 31-5
 exit code 31-5
 Expression folding 26-10
 in S-Functions 32-57
`ext_work.h` 22-11
 external mode 37-3
 architecture 37-33
 baud rates 37-17
 blocks compatible with 37-27
 client-server architecture 37-50
 command line options for target
 program 37-42
 communication channel creation 37-50
 communications overview 37-54
 configuration parameter options 37-5
 control panel options 37-13
 data archiving options 37-22
 design of 37-50
 download mechanism 37-31
`ext_comm` MEX-file
 optional arguments to (serial) 37-40
 optional arguments to (TCP/IP) 37-37
 rebuilding 37-61
 host and target systems in 37-3
 menu and toolbar items and keyboard
 shortcuts 37-8
 parameter downloading options 37-24
 Signal Viewing Subsystems in 37-28
 signals and triggering options 37-17
 target communications options 37-15
 TCP implementation 37-35
 transport layer 37-54
 using with VxWorks 34-17
 external mode API
 host source files 37-56
 implementing transport layer 37-59
 target source files 37-57
 External Target Interface dialog box
 MEX-file arguments 37-17

F

files
 for inlined S-functions 32-26
 generated. *See* generated files
 firstTime argument control, custom target support for 21-123
fixedpoint.h 22-11
float.h 22-9
 From File block
 specifying signal data file for 42-28
 Function prototype control, custom target support for 21-126
 functions
 rtw_precompile_libs 32-99
 ssSetChecksumVal 3-109

G

general code appearance options
 Maximum identifier length 20-3
 Reserved names 20-3
 generated code
 compiling and linking 27-2
 include path specification 22-19
 operations performed by 3-28
 profiling 43-7
 testing in software environment 45-2
 testing in system environment 46-2
 generated files 25-15
 contents of 22-2
 dependencies among 22-2
 model (UNIX executable) 25-17
model.c 25-15
model_capi.c 25-19
model_capi.h 25-19
model_data.c 25-17
model_dt.h 25-18
model.exe (PC executable) 25-17
model.h 25-16
model.mdl 25-15

model.mk 25-17
model_private.h 25-16
model.rtw 25-15
model_targ_data_map.m 25-18
model_target_rtw 25-20
model_types.h 25-16
modelsources.txt 25-18
rt_nonfinite.c 25-18
rt_nonfinite.h 25-18
rt_sfcn_helper.c, 25-19
rt_sfcn_helper.h 25-19
rtmodel.h 25-17
rtw_proj.tmw 25-18
rtwtypes.h 25-17
subsystem.c 25-19
subsystem.h 25-19
 generated S-functions
 tunable parameters in 3-101

H

hand-written code
 build support for 32-90
 integrating with generated code 29-2
 hardware-in-the-loop (HIL) testing
 verifying generated code in system environment with 46-2
 header files
 dependencies of
 model.h 22-8
 rtwtypes.h 22-4
 hook files
 STF_make_rtw_hook 21-23
 customizing build process with 27-22
 STF_wrap_make_cmd_hook 21-24
 hook interface
 for profiling generated code 43-7
 host
 in external mode 37-3
 host-based simulation

validating requirements with 41-1
 how to disable runtime checks
 Embedded MATLAB Coder 28-102

I

include paths
 specifying 22-19
 initial values
 tunable 12-25
 initialization
 of signals and discrete states 12-18
 inlined S-functions 32-23
 with `mdlRTW` routine 32-30
 Import block
 latch options
 generated code for option 6-6
 specifying signal data file for 42-31
 integration, code
 build support for 32-90
 intellectual property
 protecting in models 7-1
 interrupt service routine
 under VxWorks 5-4
 interrupt service routine (ISR) 21-14
 interrupt service routines 5-37
 interrupts
 handling 5-37
`intOnlyBuild` field 32-101

L

Language option
 description of 19-1
 Latch input by copying inside signal option
 generated code for 6-6
 Latch input by delaying outside signal
 generated code for 6-6
 latches
 generated code for 6-6

legacy code
 build support for 32-90
 integrating with C MEX S-functions 32-25
 integrating with generated code 29-2
 Legacy Code Tool
 deploying S-functions generated with 32-29
 generating code with 32-25
`legacy_code` function
 addressing file dependencies for code
 generation 32-28
`LibAddToCommonIncludes` function
 using for S-function build support 32-93
`LibAddToModelSources` function
 using for S-function build support 32-93
 libraries
 controlling suffix applied to names of 27-10
 model reference
 controlling the location of 27-9
 precompiled
 controlling the location of 27-8
 S-function
 precompiling 32-99
 suffixes for 32-101
 local block outputs option 26-40

M

`make` command 21-92
`make_rtw` 25-12
`makefile` 25-25
 customizations 27-7
 options for 32-101
`makefile` commands
`USE_MDLREF_LIBPATHS`
 controlling location of model reference
 libraries with 27-9
`makeInfo.precompile` `rtwmakecfg` field 32-100
`makeOpts` field 32-101
`math.h` 22-9
 MATLAB application data 21-34

mdlRTW routine
 writing inlined S-functions 32-30

MEX S-function wrapper
 definition 32-12

model (on UNIX) 25-17

Model blocks
 in Real-Time Workshop 3-51

model code
 execution of 3-32

model compiling process 22-22

model design specification
 developing 2-1

model execution
 in real time 5-11
 in Simulink 5-11
 Simulink versus real-time 5-10

Model Header block 31-3

Model Parameter Configuration dialog box
 tunable parameters and 10-1
 using 10-11

model reference
 code generation 3-51
 compatibility of top and referenced
 models 3-56
 inherited sample time and 3-66
 parameter interfacing 3-63
 project directory structure and 3-53
 signal interfacing 3-62
 subsystem code reuse and 3-73

model reference libraries
 controlling location of 27-9

Model referencing, custom target support
 for 21-110

model registration function 3-32

Model Source block 31-3

model.bat 25-18

model.c 25-15

model_capi.c 25-19

model_capi.h 25-19

model_data.c 25-17

model_dt.h 25-18

model.h 25-16

model.mdl 25-15

model.mk 25-17

model_private.h 25-16

model.rtw 25-15

model_targ_data_map.m 25-18

model_target_rtw 25-20

model_types.h 25-16

models
 checksums for 3-109
 code files for
 porting 25-29
 relocating 25-29
 integrating for real-time rapid
 prototyping 44-7
 integrating for system simulation 44-4
 protected 7-1
 protecting intellectual property in 7-1
 reference
 building in parallel 21-111

modelsources.txt 25-18

multiple models
 combining 3-114

multiport S-function example 32-23

multitasking
 automatic rate transition 5-18
 building program for 5-9
 enabling 5-9
 example model 5-27
 execution 5-30
 inserted rate transition block HTML
 report 5-20
 model execution 5-5
 operation 5-12
 task identifiers in 5-5
 task priorities 5-5
 versus single-tasking 5-3

N

noninlined S-functions 32-9
 nonvirtual subsystem code generation
 Auto option 3-41
 Function option 3-46
 Inline option 3-44
 Reusable function option 3-73
 nonvirtual subsystems
 atomic 3-37
 categories of 3-37
 conditionally executed 3-37
 modularity of code generated from 3-50

O

operating system
 tasking primitives 3-9
 optimization pane
 Stateflow and Embedded MATLAB
 options 26-46

P

parallel builds 21-111
 parameters
 interfacing 10-1
 storage declarations 10-1
 TargetPreCompLibLocation 32-100
 tunable 10-1
 tuning 10-1
 performance
 of generated code 43-7
 periodic tasks
 timers for 5-71
 persistent signals
 initialization of 12-23
 precompiled libraries
 controlling location of 27-8
 priority
 of sample rates 5-6

ProfileGenCode variable 43-9
 ProfilerTLC variable 43-9
 program architecture
 embedded 3-34
 initialization functions 3-24
 main function 3-24
 model execution 3-25
 program execution 3-13
 program termination 3-25
 program timing 3-11
 rapid prototyping 3-22
 real-time 3-22
 termination functions 3-32
 project directory 25-20
 prototyping, rapid
 integrating component models for 44-7
 pseudomultitasking 5-6

R

rapid prototyping
 integrating component models for 44-7
 rapid simulation target 42-2
 batch simulations (Monte Carlo) 42-23
 command line options 42-24
 limitations 42-37
 output filename specification 42-36
 parameter structure access 42-28
 signal data file specification for From File
 block 42-28
 signal data file specification for Import
 block 42-31
 rate transition block
 and continuous sample time 5-20
 Rate Transition block 5-13
 auto-insertion of 5-18

real time
 executing models in 5-11
 integrating continuous states in 3-26

real-time malloc target 18-16
 combining models with 3-114

Real-time model
 description 3-29

real-time model data structure 18-18

Real-Time Workshop
 parameters
 interfacing 10-1
 storage 10-1
 tuning 10-1
 specifying nonvirtual code generation 3-37
 third-party compilers
 configuration A-1
 support 25-2

Real-Time Workshop pane 16-2
 Language option 19-1
 opening 16-2
 target configuration options
 Browse button 18-29
 Generate makefile 25-11
 make command field 25-12
 system target file field 18-29
 template makefile field 25-12
 TLC options 25-11
 recommended target features 21-5
 reference models
 building in parallel 21-111
 referenced models
 code generation incompatibilities 3-56
 generating code for 3-51

registration files
 multiple
 for code generation 32-28

requirements
 validating with host-based simulation 41-1

reset value

initial value as 12-24

root models
 Custom Code blocks in 31-3

rsim
 See rapid simulation target 42-2

rt_logging.h 22-11

rt_nonfinite.c 25-18

rt_nonfinite.h 25-18

rt_sfcn_helper.c 25-19

rt_sfcn_helper.h 25-19

rtm macros 18-18

rtModel 18-18

rtmodel.h 25-17

rtw_continuous.h 22-12

rtw_extmode.h 22-12

rtw_local_blk_outs 11-7

rtw_matlogging.h 22-12

rtw_precompile_libs function 32-99

rtw_proj.tmw 25-18

rtw_solver.h 22-12

RTWdata structure
 Inlining an S-function 32-31

rtwgensettings structure 21-45

rtwlib command 31-2

rtwmakecfg field
 TargetPreCompLibLocation 32-100

rtwmakecfg.m
 creating S-functions 32-94
 generating for C MEX S-functions 32-28
 using for S-functions 32-93

rtwmakecfgDirs field 32-101

rtwMakecftDirs field 32-101

rtwoptions structure
 callbacks in 21-53
 example of 21-49
 fields in 21-51
 overview of 21-48

rtwtypes.h 25-17

S

S-Function blocks
 masked
 configured to call existing external code 32-25

S-function libraries
 precompiling 32-99

S-function target 18-17
 applications of 3-93
 automatic S-function generation 3-104
 generating reusable components with 3-95
 tunable parameters in 3-101

S-Function target
 checksums and 3-109

S-functions
 API 3-32
 build support for 32-90
 creating `rtwmakecfg.m` for 32-94
 deploying generated 32-29
 fully inlined with `mdlRTW` routine 32-30
 generating automatically 3-104
 implicit build support for 32-91
 inlined 32-23
 generated with Legacy Code Tool 32-25
 generating files for 32-26

models containing 3-32

modifying TMF for 32-97

noninlined 3-32 32-9

setting `SFunctionModules` parameter
 for 32-91

that work with Real-Time Workshop 32-2

types of 32-4

using `rtwmakecfg.m` for 32-93
 using TLC library functions for 32-93
 wrapper 32-12

sample rate transitions 5-13
 faster to slower
 in real-time 5-21
 in Simulink 5-21
 slower to faster

in real-time 5-24
 in Simulink 5-23

sample time constraints
 setting for multitasking 5-5

sample time properties
 setting for multitasking 5-5

`SFunctionModules` parameter
 setting 32-91

signal data
 specifying for From File block 42-28
 specifying for Import block 42-31

signal initialization
 in generated code 12-23

signal objects
 initializing 12-18

signal properties 10-31
 setting by using Signal Properties dialog box 10-31

signal storage reuse option 26-39

Signal Viewing Subsystems 37-28

signals
 initializing 12-18

`simstruc.h` 22-13

`simstruc_types.h` 22-13

`SimStruct` data structure
 and global registration function 3-32
 definition of 3-28

simulation
 integrating component models for 44-4
 validating requirements with 41-1

Simulink
 and Real-Time Workshop
 block execution order 6-7
 interactions to consider 6-1
 sample time propagation 6-4
 using data objects 12-1
 using parameter objects 12-4
 using signal objects 12-12

Simulink data objects 12-1
 parameter objects 12-4

- signal objects 12-12
- single-tasking 5-9
 - building program for 5-9
 - enabling 5-10
 - example model 5-27
 - execution 5-28
 - operation 5-12
- s1prj** directory 25-20
- software environment
 - testing generated code in 45-2
- software-in-the-loop (SIL) testing
 - verifying generated code in software environment with 45-2
- specification
 - developing 2-1
- ssSetChecksumVal** function 3-109
- Start button menu
 - info.xml file for 21-27
- states, discrete
 - initializing 12-18
- <stddef.h>** 22-10
- <stdio.h>** 22-10
- <stdlib.h>** 22-10
- step size
 - of real-time continuous system 3-27
- StethoScope
 - See* VxWorks 34-16
- STF_make_rtw_hook** function
 - arguments to 27-23
- storage classes
 - required for signal initialization 12-18
- <string.h>** 22-11
- subsystem
 - nonvirtual 3-37
- subsystem.c** 25-19
- subsystem.h** 25-19
- subsystems
 - checksums for 3-109
 - custom code blocks in 31-9
- suffixes
 - precompiled library 32-101
 - <sysran_types.h>** 22-13
- System Derivatives function block 31-5
- System Disable function block 31-4
- System Enable function block 31-4
- system environment
 - testing generated code in 46-2
- System Initialize function block 31-4
- System Outputs function block 31-4
- system simulation
 - integrating component models for 44-4
- System Start function block 31-4
- system target file 22-23
- system target file (STF)
 - customization techniques 21-56
 - defining target options in 21-44
 - header comments section 21-41
 - location of 21-38
 - naming conventions for 21-38
 - overview of 21-37
 - Release 14 or later compatibility issues
 - callback conversion API 21-65
 - rtwoptions callbacks 21-64
 - target options display 21-69
 - RTW_OPTIONS** section 21-44
 - rtwgensettings** structure 21-45
 - structure of 21-38
 - target options inheritance mechanism 21-55
 - TLC entry point in 21-43
 - TLC variables section 21-42
- System Target File Browser 18-29
- system target file creation
 - tutorial 21-71
- system target files
 - selecting programmatically 18-31
- System Terminate function block 31-4
- System Update function block 31-4

T

target
 rapid simulation
See rapid simulation target 42-2
 real-time malloc
See real-time malloc target 18-16
 target directories
 blocks Folder 21-18
 central folder 21-18
 development tool support files in 21-20
 for common source files 21-20
 for target preferences classes 21-20
 naming conventions 21-11
 target root 21-12
 target root folder 21-17
 target file
 system 22-23
 target files
 main.c 21-23
 naming conventions 21-11
 system target file (STF) 21-21
 target settings file 21-22
 template makefile (TMF) 21-22
 Target function library option
 relationship to TGT_FCN_LIB variable 27-3
 Target Language Compiler
 code generation variables 21-42
 function library 22-22
 generation of code by 22-22
 TLC scripts 22-22
 target options inheritance 21-55
 mechanism for 21-55
 target root folder 21-12
 target types
 baseline 21-3
 cosimulation 21-4
 turnkey 21-4
 TargetLibSuffix parameter
 controlling suffix applied to library names
 with 27-10

TargetPreCompLibLocation parameter 32-100
 controlling location of precompiled libraries
 with 27-8
 targets
 available configurations 18-24
 selecting programmatically 18-31
 task
 spawning 5-45
 task identifier (tid) 5-5
 template makefile
 compiler-specific 18-32
 structure of 21-86
 tokens 21-87
 template makefile options
 LCC 18-38
 UNIX 18-34
 Visual C++ 18-35
 Watcom 18-37
 template makefile variables
 TGT_FCN_LIB 27-3
 TGT_FCN_LIB template makefile variable 27-3
 time counters 5-70
 in triggered subsystems 5-72
 timer, elapsed
 example 5-77
 timers 5-70
 allocation of 5-71
 APIs for accessing 5-72
 integer 5-72
 timing services 5-70
 TLC API
 for code generation 5-76
 TLC block file
 generating for C MEX S-functions 32-26
 TLC hook function interface
 for profiling generated code 43-7
 TLC library functions
 using for inlined S-functions 32-93
 TMFs
 modifying for S-functions 32-97

tokens 21-87
T
 Tornado
See VxWorks 34-1
 tunable expressions 10-1 10-16
 in masked subsystems 10-16
 operators, restrictions on 10-18
 tunable parameters
 in signal initial values 12-25
 tutorials
 creating custom target configuration 21-71

U

USE_MDLREF_LIBPATHS command
 controlling location of model reference
 libraries with 27-9
 user code
 build support for 32-90
 integrating with generated code 29-2

V

Variables
 ProfileGenCode 43-9
 ProfilerTLC 43-9
VxWorks
 application
 building 34-22
 application architecture 34-4
 application implementation 34-12
 automatic download 34-22
 specifying 34-17
 automatic downloading
 configuring for 34-21
 automatic execution 34-22
 base task priority
 specifying 34-17
 block library 34-3
 code generation options
 specifying 34-13

communication tasks 34-8
 configuration 34-10
 verifying 34-11
 CPU
 specifying 34-19
 device driver blocks
 adding 34-12
 execution tasks 34-7
 external mode tasks 34-9
 files
 downloading 34-24
 generated code format
 specifying 34-16
 GNU tools for 34-19
 hardware architecture 34-4
 header file paths
 resolving 34-20
 host processes 34-6
 installation 34-10
 manual download 34-24
 manual execution 34-24
 MAT-file logging
 specifying 34-15
 module architecture 34-6
 multitasking 34-8
 optStr
 passing 34-27
 program
 starting 34-25
 resetting 34-24
 resources for applications 34-2
 rt_main
 calling 34-27
 Simulink external mode 34-17
 Simulink model
 designing 34-12
 single-tasking 34-7
 software architecture 34-5
 solver
 specifying 34-14

- StethoScope monitoring
 - specifying 34-16
- StethoScope tasks 34-8
- target 34-4
 - connecting to 34-10
 - downloading to 34-24
- target type
 - specifying 34-19
- target-specific options
 - setting 34-15
- task priority 34-8
- task stack size
 - specifying 34-17
- tasks created by 34-7
- template makefile 34-19
- configuring 34-19
- tool locations 34-19
- Tornado environment 34-2
- Tornado target 34-3
 - selecting 34-13
- vxlib** library obsolete 34-3
- VxWorks task
 - spawning 5-45

W

- when to disable runtime checks
- Embedded MATLAB Coder 28-102
- working directory 25-19
- wrapper S-functions 32-12