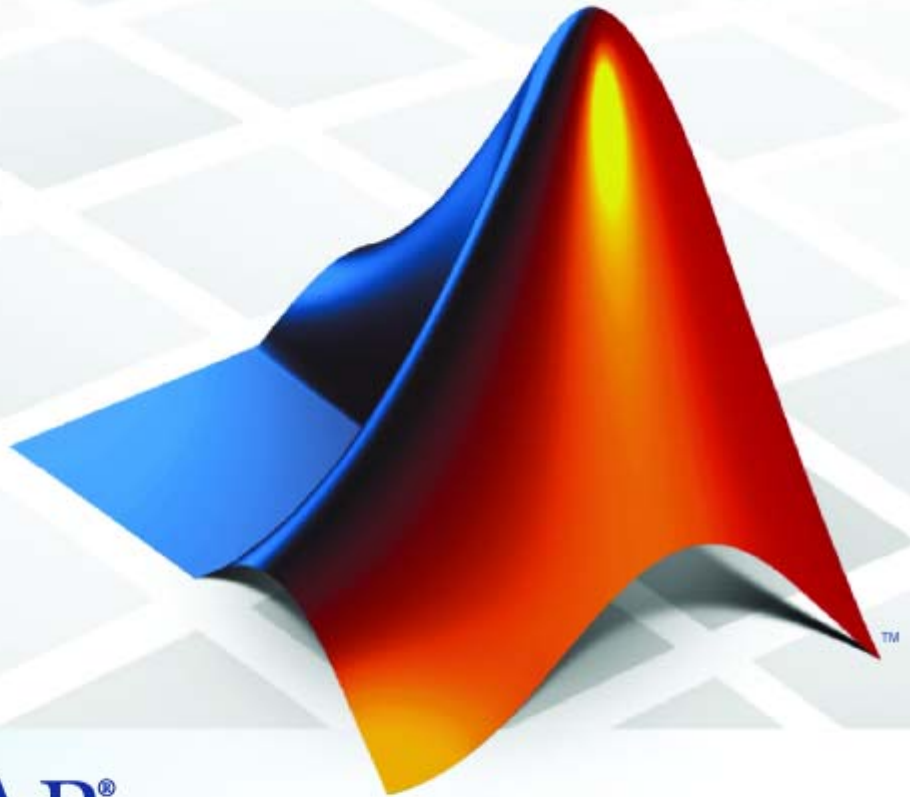


Real-Time Workshop[®] Embedded Coder[™] 5

User's Guide



MATLAB[®]
& **SIMULINK[®]**

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® Embedded Coder™ User's Guide

© COPYRIGHT 2002–2010 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

July 2002	Online only	Version 3.0 (Release 13)
December 2003	Online only	Revised for Version 3.2 (Release 13SP1+)
June 2004	Online only	Revised for Version 4.0 (Release 14)
October 2004	Online only	Revised for Version 4.1 (Release 14SP1)
March 2005	Online only	Revised for Version 4.2 (Release 14SP2)
September 2005	Online only	Revised for Version 4.3 (Release 14SP3)
March 2006	Online only	Revised for Version 4.4 (Release 2006a)
September 2006	Online only	Revised for Version 4.5 (Release 2006b)
March 2007	Online only	Revised for Version 4.6 (Release 2007a)
September 2007	Online only	Revised for Version 5.0 (Release 2007b)
March 2008	Online only	Revised for Version 5.1 (Release 2008a)
October 2008	Online only	Revised for Version 5.2 (Release 2008b)
March 2009	Online only	Revised for Version 5.3 (Release 2009a)
September 2009	Online only	Revised for Version 5.4 (Release 2009b)
March 2010	Online only	Revised for Version 5.5 (Release 2010a)

Introduction to the Real-Time Workshop® Embedded Coder Product

1

Developing Models for Code Generation

Setting Up Your Modeling Environment

2

Architecture Considerations

3

Generating Code Variants for Variant Models	3-2
Prerequisites	3-2
Why Generate Code Variants?	3-2
How to Generate Code Variations for Model Variants	3-3
Reviewing Code Variants in the Code Generation	
Report	3-5
Example of Variants in the Generated Code	3-6
Demo for Code Variants Using Model Blocks	3-8
Creating and Using Host-Based Shared Libraries	3-9
Overview	3-9
Generating a Shared Library Version of Your Model	
Code	3-10
Creating Application Code to Load and Use Your Shared	
Library File	3-11
Host-Based Shared Library Limitations	3-15

Scheduling Considerations

4

Using Discrete and Continuous Time	4-2
Generating Code for Discrete and Continuous Time	
Blocks	4-2
Generating Code that Supports Continuous Solvers	4-2
Generating Code that Honors a Stop Time	4-3
Optimizing Task Scheduling for Multirate Multitasking	
Models on RTOS Targets	4-4
Overview	4-4
Using rtmStepTask	4-5
Task Scheduling Code for Multirate Multitasking Model on	
Wind River Systems VxWorks Target	4-5
Suppressing Redundant Scheduling Calls	4-6

Developing Model Patterns that Generate Specific C Constructs

5

About Modeling Patterns	5-2
Standard Methods to Prepare a Model for Code	
Generation	5-3
Configuring a Signal	5-3
Configuring Input and Output Ports	5-4
Initializing States	5-4
Setting Up Configuration Parameters for Code	
Generation	5-4
Setting Up an Example Model With a Stateflow Chart ...	5-5
Setting Up an Example Model With an Embedded MATLAB	
Block	5-7
Types, Operators, and Expressions	5-8
Data Declaration	5-8
Data Type Conversion	5-11
Type Qualifiers	5-15

Relational and Logical Operators	5-17
Bitwise Operations	5-21
Control Flow	5-25
If-Else	5-25
Switch	5-32
For loop	5-38
While loop	5-46
Do While loop	5-57
Functions	5-64
Function Call	5-64
Function Prototyping	5-66
External C Functions	5-69

Defining Data Representation and Storage for Code Generation

Using mpt Data Objects

6

Creating and Using Custom Storage Classes

7

Introduction to Custom Storage Classes	7-2
Custom Storage Class Memory Sections	7-3
Registering Custom Storage Classes	7-3
Custom Storage Class Demos	7-4
Resources for Defining Custom Storage Classes	7-5
Simulink Package Custom Storage Classes	7-6
Creating Packages that Support CSC Definitions	7-8

Designing Custom Storage Classes and Memory	
Sections	7-12
Using the Custom Storage Class Designer	7-12
Editing Custom Storage Class Properties	7-19
Using Custom Storage Class References	7-26
Creating and Editing Memory Section Definitions	7-31
Using Memory Section References	7-34
Applying CSCs to Parameters and Signals	7-37
About Applying Custom Storage Classes	7-37
Applying a Custom Storage Class to a Parameter	7-38
Applying a Custom Storage Class to a Signal	7-40
Applying a CSC Using a Base Workspace Signal Object ..	7-41
Applying a CSC Using an Embedded Signal Object	7-43
Specifying a Custom Storage Class Using the GUI	7-50
Specifying a Custom Storage Class Using the API	7-53
Generating Code with Custom Storage Classes	7-58
Code Generation Prerequisites	7-58
Code Generation Example	7-58
Defining Advanced Custom Storage Class Types	7-62
Introduction	7-62
Create Your Own Parameter and Signal Classes	7-62
Create a Custom Attributes Class for Your CSC (Optional)	7-62
Write TLC Code for Your CSC	7-63
Register Custom Storage Class Definitions	7-63
GetSet Custom Storage Class for Data Store Memory ..	7-66
Overview	7-66
GetSet CSC Properties	7-66
Using the GetSet CSC	7-67
GetSet CSC Restrictions	7-67
GetSet Custom Storage Class Example	7-68
Custom Storage Class Implementation	7-70
Custom Storage Class Limitations	7-72
Custom Storage Classes Prior to R2009a	7-73

Custom Storage Classes Prior to Release 14	7-74
Introduction	7-74
Simulink.CustomParameter Class	7-74
Simulink.CustomSignal Class	7-75
Instance-Specific Attributes for Older Storage Classes ...	7-78
Assigning a Custom Storage Class to Data	7-79
Code Generation with Older Custom Storage Classes	7-80
Compatibility Issues for Older Custom Storage Classes ..	7-81

Inserting Comments and Pragmas in Generated Code

8

Introduction to Memory Sections	8-2
Overview	8-2
Memory Sections Demo	8-2
Additional Information	8-2
 Requirements for Defining Memory Sections	 8-4
 Defining Memory Sections	 8-7
Editing Memory Section Properties	8-7
Specifying the Memory Section Name	8-9
Specifying a Qualifier for Custom Storage Class Data Definitions	8-9
Specifying Comment and Pragma Text	8-9
Surrounding Individual Definitions with Pragmas	8-9
Including Identifier Names in Pragmas	8-10
 Configuring Memory Sections	 8-11
 Applying Memory Sections	 8-12
Assigning Memory Sections to Custom Storage Classes ..	8-12
Applying Memory Sections to Model-Level Functions and Internal Data	8-14
Applying Memory Sections to Atomic Subsystems	8-16
 Examples of Generated Code with Memory Sections ..	 8-20

Sample ERT-Based Model with Subsystem	8-20
Model-Level Data Structures	8-22
Model-Level Functions	8-22
Subsystem Function	8-23
Memory Section Limitation	8-25

Optimizing Buses for Code Generation

9

Introduction	9-2
Setting Bus Diagnostics	9-3
Optimizing Virtual and Nonvirtual Buses	9-4
Use Virtual Buses Wherever Possible	9-4
Avoid Nonlocal Nested Buses in Nonvirtual Buses	9-5
Using Single-Rate and Multi-Rate Buses	9-7
Introduction	9-7
Techniques for Combining Multiple Rates	9-7
Larger Buses and Multiple Rates	9-9
Specifying Sample Time Rates	9-10
Setting Bus Signal Initial Values	9-11
Introduction	9-11
Initializing Bus Signals in Simulink	9-11
Bus Initialization in Stateflow and Embedded MATLAB ..	9-12
Creating a Bus of Constants	9-14
Buses and Atomic Subsystems	9-16
Extract Nonvirtual Bus Signals Inside of Atomic	
Subsystems	9-16
Virtual Bus Signals Crossing Atomic Boundaries	9-17
Atomic Subsystems and Buses of Constants	9-19

Renaming and Replacing Data Types

10

Defining Application-Specific Data Types Based On Built-In Types	10-2
Code Generation with User-Defined Data Types	10-4
Overview	10-4
Specifying Type Definition Location for User-Defined Data Types	10-5
Using User-Defined Data Types for Code Generation	10-6

Managing Data Definitions and Declarations With the Data Dictionary

11

Overview of the Data Dictionary	11-2
Creating Simulink and mpt Data Objects	11-4
Overview	11-4
Creating Simulink Data Objects with Data Object Wizard	11-5
Creating mpt Data Objects with Data Object Wizard	11-11
Comparing Simulink and mpt Data Objects	11-12
Creating Data Objects Based on an External Data Dictionary	11-16
Creating a Data Dictionary for a Model	11-19
Using Data Object Wizard	11-19
Inspect the Data Dictionary	11-23
Generate and Inspect Code	11-24
Defining All Global Data Objects in a Separate File ...	11-26
Defining a Specific Global Data Object in Its Own File	11-28

Saving and Loading Data Objects	11-29
Applying Naming Rules to Identifiers Globally	11-30
Overview	11-30
Changing Names of Identifiers	11-31
Specifying Simulink Data Object Naming Rules	11-34
Defining Rules That Change All Signal Names	11-35
Defining Rules That Change All Parameter Names	11-35
Defining Rules That Change All #defines	11-36
Creating User Data Types	11-38
Overview	11-38
Registering User Data Types Using <code>sl_customization.m</code> ..	11-39
Example User Data Type Customization Using <code>sl_customization.m</code>	11-41
Selecting User Data Types for Signals and Parameters	11-43
Preparing User Data Types	11-43
Selecting the User Data Types	11-45
Registering mpt User Object Types	11-48
Introduction	11-48
Registering mpt User Object Types Using <code>sl_customization.m</code>	11-48
Example mpt User Object Type Customization Using <code>sl_customization.m</code>	11-50
Replacing Built-In Data Type Names in Generated Code	11-52
Replacing Built-In Data Type Names	11-52
Replacing <code>boolean</code> with an Integer Data Type	11-58
Data Type Replacement Limitations	11-60
Customizing Data Object Wizard User Packages	11-61
Introduction	11-61
Registering Data Object Wizard User Packages Using <code>sl_customization.m</code>	11-61
Example Data Object Wizard User Package Customization Using <code>sl_customization.m</code>	11-63

Managing Placement of Data Definitions and Declarations

12

Overview of Data Placement	12-2
Priority and Usage	12-3
Overview	12-3
Read-Write Priority	12-5
Global Priority	12-7
Definition File, Header File, and Ownership Priorities ...	12-9
Ownership Settings	12-10
Memory Section Settings	12-11
Data Placement Rules	12-12
Example Settings	12-13
Introduction	12-13
Read-Write Example	12-15
Ownership Example	12-17
Header File Example	12-18
Definition File Example	12-20
Data Placement Rules and Effects	12-22
Effects of Ownership Settings	12-22
Example Settings and Resulting Generated Files	12-23
Data Placement Rules	12-25

Specifying the Persistence Level for Signals and Parameters

13

Preparing Models for Code Generation

Mapping Application Objectives to Model Configuration Parameters

14

Considerations When Mapping Application Objectives	14-2
Defining High-Level Code Generation Objectives	14-3
Determining Whether the Model is Configured for Specified Objectives	14-4
Specifying Code Generation Objectives Using the GUI ...	14-4
Specifying Code Generation Objectives at the Command Line	14-6
Reviewing Objectives in Referenced Models	14-7
Reviewing the Model Without Generating Code	14-7
Reviewing the Model During Code Generation	14-9
Creating Custom Objectives	14-11
Specifying Parameters in Custom Objectives	14-11
Specifying Checks in Custom Objectives	14-12
Determining Checks and Parameters in Existing Objectives	14-12
How to Create Custom Objectives	14-14

Choosing and Configuring an Embedded Real-Time Target

15

Introduction	15-2
Selecting an ERT Target	15-4
Selecting a System Target File	15-5
Selecting a System Target File Programmatically	15-7
Customizing an ERT Target	15-8

Specifying Code Appearance and Documentation

16

Customizing Comments in Generated Code	16-2
Adding Custom Comments to Generated Code	16-2
Adding Global Comments	16-5
Configuring the Appearance of Generated Identifiers	16-11
Customizing Generated Identifiers	16-11
Configuring Symbols	16-12
Controlling Code Style	16-21
Configuring Templates for Customizing Code Organization and Format	16-22
Overview	16-23
Custom File Processing Components	16-24
Custom File Processing User Interface Options	16-24
Code Generation Template (CGT) Files	16-26
Using Custom File Processing (CFP) Templates	16-30

Custom File Processing (CFP) Template Structure	16-30
Changing the Organization of a Generated File	16-32
Generating Source and Header Files with a Custom File Processing (CFP) Template	16-34
Comparison of a Template and Its Generated File	16-43
Code Template API Summary	16-46
Generating Custom File and Function Banners	16-49
Template Symbols and Rules	16-58

Configuring the Placement of Data in Generated Code	16-67
--	--------------

Ensuring Delimiter Is Specified for All #Includes	16-68
--	--------------

Defining Model Configuration Variations

17

Introduction	17-2
---------------------------	-------------

Viewing ERT Target Options in the Configuration Parameters Dialog Box or Model Explorer	17-3
--	-------------

Generating Code and Building Executables

Generating Code Modules

18

Code Modules	18-2
Introduction	18-2
Generated Code Modules	18-2
User-Written Code Modules	18-5
Customizing Generated Code Modules	18-5

Generating Reports for Code Reviews and Traceability Analysis

19

About HTML Code Generation Report Extensions	19-2
Generating an HTML Code Generation Report	19-4
Using the Code Interface Report to Analyze the	
Generated Code Interface	19-6
Code Interface Report Overview	19-6
Generating a Code Interface Report	19-7
Navigating Code Interface Report Subsections	19-9
Interpreting the Entry Point Functions Subsection	19-10
Interpreting the Inports and Outports Subsections	19-13
Interpreting the Interface Parameters Subsection	19-14
Interpreting the Data Stores Subsection	19-16
Code Interface Report Limitations	19-17

Optimizing Generated Code

20

Configuring Production Code Optimizations	20-2
Optimization Dependencies	20-4
Optimizing Your Model with Configuration Wizard	
Blocks and Scripts	20-6
Overview	20-6
Adding a Configuration Wizard Block to Your Model	20-8
Using Configuration Wizard Blocks	20-11
Creating a Custom Configuration Wizard Block	20-11
Tips for Optimizing the Generated Code	20-19
Introduction	20-19
Using Configuration Wizard Blocks	20-19

Setting Hardware Implementation Parameters	
Correctly	20-20
Removing Unnecessary Initialization Code	20-22
Generating Pure Integer Code If Possible	20-23
Disabling MAT-File Logging	20-23
Using Virtualized Output Ports Optimization	20-24
Controlling Signal Storage	20-25
Using External Mode with the ERT Target	20-26

Developing Models and Code That Comply with Industry Standards and Guidelines

21

What Are the Standards and Guidelines?	21-2
Developing Models and Code That Comply with MAAB Guidelines	21-4
Developing Models and Code That Comply with MISRA C Guidelines	21-5
Developing Models and Code That Comply with the IEC 61508 Standard	21-6
Applying Simulink and Real-Time Workshop Technology to the IEC 61508 Standard	21-6
Checking for IEC 61508 Standard Compliance Using the Model Advisor	21-6
Validating Traceability	21-7
Developing Models and Code That Comply with the DO-178B Standard	21-8
Applying Simulink and Real-Time Workshop Technology to the DO-178B Standard	21-8
Checking for Standard Compliance Using the Model Advisor	21-8
Validating Traceability	21-9

Generating Code for AUTOSAR Software Components

22

Overview of AUTOSAR Support	22-2
Simulink Modeling Patterns for AUTOSAR	22-3
AUTOSAR Software Components	22-3
AUTOSAR Communication	22-11
Calibration Parameters	22-19
Inter-Runnable Variables	22-19
Data Types	22-20
AUTOSAR Terminology	22-25
Workflow for AUTOSAR	22-28
Importing an AUTOSAR Software Component	22-30
Using the Configure AUTOSAR Interface Dialog Box ..	22-32
Configuring Ports for Basic Software and Error Status	
Receivers	22-37
Configuring Client-Server Communication	22-39
Configuring a Server Operation	22-40
Configuring the Invoke AUTOSAR Server Operation	
Block	22-43
Creating Configurable Subsystems from a Client-Server	
Interface	22-45
Simulating and Generating Code for Client-Server	
Communication	22-46
Configuring Calibration Parameters	22-48
Modifying and Validating an Existing AUTOSAR	
Interface	22-49
Exporting a Single Runnable AUTOSAR Software	
Component	22-50

Selecting an AUTOSAR Schema	22-51
Generating AUTOSAR Compiler Abstraction Macros	22-52
Configuring Multiple Runnables	22-55
Configuring AUTOSAR Options Programmatically ...	22-58
Verifying the AUTOSAR Code with SIL and PIL	
Simulations	22-59
Overview	22-59
Using the SIL and PIL Simulation Modes	22-59
Using a SIL Block for AUTOSAR Verification	22-60
Limitations and Tips	22-62
Cannot Import Internal Behavior	22-62
Cannot Copy Subsystem Blocks Without Losing Interface Information	22-62
Error If No Default Configuration	22-63
Server Operation Model with Reusable Function Subsystems	22-63
Invoke AUTOSAR Server Operation Block in Referenced Model	22-63
Server Operation and Model Block with States	22-63
Cannot Save Importer Objects in MAT-Files	22-63
Using the Merge Block for Inter-Runnable Variables	22-63
AUTOSAR Compiler Abstraction Macros	22-64
Intrinsic Fixed-Point Types for Model Configured as Server	22-64
Server Operation Model with Tunable Parameters	22-65
Migrating AUTOSAR Development Kit Models	22-65
Demos and Further Reading	22-66
AUTOSAR Demos	22-66
Further Reading	22-67

Integrating External Code and Generated C and C++ Code

About External Code Integration Extensions

23

Generating S-Function Wrappers

24

About S-Function Wrapper Generation	24-2
Creating a SIL Block	24-4
S-Function Wrapper Generation Limitations	24-6

Exporting Function-Call Subsystems

25

Overview	25-2
Exported Subsystems Demo	25-3
Additional Information	25-3
Requirements for Exporting Function-Call Subsystems	25-4
Requirements for All Exported Subsystems	25-4
Requirements for Exported Virtual Subsystems	25-5
Techniques for Exporting Function-Call Subsystems ..	25-7
Optimizing Exported Function-Call Subsystems	25-8

Exporting Function-Call Subsystems That Depend on Elapsed Time	25-9
Function-Call Subsystem Export Example	25-10
Function-Call Subsystems Export Limitations	25-14

Nonvirtual Subsystem Modular Function Code Generation

26

Overview	26-2
Configuring Nonvirtual Subsystems for Generating Modular Function Code	26-4
Examples of Modular Function Code for Nonvirtual Subsystems	26-9
H File Differences for Nonvirtual Subsystem Function Data Separation	26-11
C File Differences for Nonvirtual Subsystem Function Data Separation	26-13
Nonvirtual Subsystem Modular Function Code Limitations	26-15

Controlling Generation of Function Prototypes

27

Overview	27-2
Configuring Model Function Prototypes	27-4
Launching the Model Interface Dialog Boxes	27-4
Default Model Initialize and Step Functions View	27-4

Model Specific C Prototypes View	27-5
Configuring Function Prototypes for Nonvirtual Subsystems	27-9
Model Function Prototypes Example	27-12
Configuring Model Function Prototypes Programmatically	27-18
Sample Script for Configuring Model Function Prototypes	27-22
Verifying Generated Code for Customized Functions ..	27-23
Model Function Prototype Control Limitations	27-24

Controlling Generation of Encapsulated C++ Model Interfaces

28

Overview of C++ Encapsulation	28-2
C++ Encapsulation Quick-Start Example	28-4
Generating and Configuring C++ Encapsulation	
Interfaces to Model Code	28-12
Selecting the C++ (Encapsulated) Option	28-12
Configuring Code Interface Options	28-13
Configuring the Step Method for Your Model Class	28-15
Configuring C++ Encapsulation Interfaces for Nonvirtual Subsystems	28-20
Configuring C++ Encapsulation Interfaces	
Programmatically	28-22

Sample Script for Configuring the Step Method for a Model Class	28-25
C++ Encapsulation Interface Control Limitations	28-27

Replacing Math Functions and Operators Using Target Function Libraries

29

Introduction to Target Function Libraries	29-2
Overview of Target Function Libraries	29-2
Target Function Libraries General Workflow	29-7
Target Function Libraries Quick-Start Example	29-9
Creating Function Replacement Tables	29-16
Overview of Function Replacement Table Creation	29-16
Creating Table Entries	29-20
Example: Mapping Math Functions to Target-Specific Implementations	29-27
Example: Mapping the memcpy Function to a Target-Specific Implementation	29-34
Example: Mapping Nonfinite Support Utility Functions to Target-Specific Implementations	29-39
Example: Mapping Scalar Operators to Target-Specific Implementations	29-44
Mapping Nonscalar Operators to Target-Specific Implementations	29-50
Mapping Fixed-Point Operators to Target-Specific Implementations	29-80
Remapping Operator Outputs to Implementation Function Input Positions	29-115
Refining TFL Matching and Replacement Using Custom TFL Table Entries	29-117
Specifying Build Information for Function Replacements	29-134
Adding Target Function Library Reserved Identifiers	29-137
Examining and Validating Function Replacement Tables	29-140

Overview of Function Replacement Table Validation	29-140
Invoking the Table Definition File	29-140
Using the Target Function Library Viewer to Examine Your Table	29-141
Using the Target Function Library Viewer to Examine Registered TFLs	29-142
Tracing Code Generated Using Your Target Function Library	29-144
Examining TFL Cache Hits and Misses	29-145
Registering Target Function Libraries	29-149
Overview of TFL Registration	29-149
Using the sl_customization API to Register a TFL with Simulink Software	29-150
Using the rtwTargetInfo API to Register a TFL with Embedded MATLAB Coder Software	29-154
Registering Multiple TFLs	29-156
Target Function Library Limitations	29-158

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

Configuring the Target Hardware Environment

30

Configuring Support for Numeric Data	30-2
Configuring Support for Time Values	30-3
Setting Up Support for Non-Inlined S-Functions	30-4
Configuring Model Function Generation and Argument Passing	30-5

Setting Up Support for Code Reuse	30-7
Configuring Target Function Libraries	30-8

Interfacing With Hardware That is Not Running an Operating System (Bare Board)

31

About Standalone Program Execution	31-2
Generating a Standalone Program	31-3
Standalone Program Components	31-4
Main Program	31-5
Overview of Operation	31-5
Guidelines for Modifying the Main Program	31-5
rt_OneStep and Scheduling Considerations	31-7
Overview of Operation	31-7
Single-Rate Single-tasking Operation	31-8
Multirate Multitasking Operation	31-9
Multirate Single-Tasking Operation	31-12
Guidelines for Modifying rt_OneStep	31-12
Model Entry Points	31-14
Static Main Program Module	31-16
Overview	31-16
Rate Grouping and the Static Main Program	31-17
Modifying the Static Main Program	31-18
Rate Grouping Compliance and Compatibility	
Issues	31-21
Main Program Compatibility	31-21
Making Your S-Functions Rate Grouping Compliant	31-21

Wind River Systems VxWorks Example Main Program

32

Introduction to the VxWorks Example Main Program	32-2
Task Management	32-3
Overview of Operation	32-3
Single-Rate Single-tasking Operation	32-3
Multirate Multitasking Operation	32-4
Multirate Single-tasking Operation	32-4

Verifying Generated Code Applications

Tracing Generated Code to Requirements

33

About Generated Code and Requirements	
Traceability	33-2
Goals of Generated Code and Requirements	
Traceability	33-3

Verifying Generated Code

34

Traceability for Production Code Generation	34-2
About Traceability	34-2
Tracing Code to Model Objects Using Hyperlinks	34-2
Tracing Model Objects to Generated Code	34-4
Reloading Existing Traceability Information	34-6
Customizing Traceability Reports	34-7
Traceability Limitations	34-9

Checking Code Correctness	34-11
About Checking Code Correctness	34-11
How To Check Code Correctness	34-11
PolySpace Analysis Options	34-12

Rapid Prototyping On a Target System

35

About On-Target Rapid Prototyping	35-2
Goals of On-Target Rapid Prototyping	35-3
Optimizing Generated Code for an Embedded Processor With On-Target Rapid Prototyping	35-4

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

36

About Software-in-the-Loop Simulation	36-2
Verifying Code Using the SIL Simulation Mode	36-3
About SIL Simulation Mode	36-3
Comparison of SIL and PIL Simulation	36-3
Choosing a SIL Approach	36-5
About Choosing a SIL Simulation	36-5
When to Use Top-Model SIL	36-5
When to Use Model Block SIL	36-6
When to Use the SIL Block	36-10
Running a Complete Model as a SIL Simulation	36-12
Running a Referenced Model as a SIL Simulation	36-14

Verifying Internal Signals of a Component	36-14
Simulation Mode Override Behavior in Model Reference Hierarchy	36-15
SIL Code Interfaces	36-16
Code Interface for Top-Model SIL	36-16
Code Interface for Model Block SIL	36-16
Configuring a SIL Simulation	36-18
Configuring a Top-Model SIL Simulation	36-18
Configuring a Model Block SIL Simulation	36-20
Using a Code Coverage Tool in a SIL Simulation	36-22
Verifying a SIL or PIL Configuration	36-26
Compatible Models	36-28
Verifying Code Using a SIL Block	36-30
Configuring Hardware Implementation Settings	36-31
Compiling Generated Code That Supports Portable Word Sizes	36-33
Portable Word Sizes Limitation	36-33
SIL Support and Limitations	36-34

Verifying a Component in the Target Environment

37

About Component Verification in the Target Environment	37-2
Goals of Component Verification in the Target Environment	37-3
Maximizing Code Portability and Configurability	37-4

Simplifying Code Integration and Maximizing Code Efficiency	37-5
Running Component Tests in the Target Environment	37-7

Verifying a Component by Building a Complete Real-Time Target Environment

38

About Component Verification With a Complete Real-Time Target Environment	38-2
Goals of Component Verification With a Complete Real-Time Target Environment	38-4
Testing a Component as Part of a Complete Real-Time Target Environment	38-5

Verifying Compiled Object Code with Processor-in-the-Loop Simulation

39

About Processor-In-the-Loop Simulation	39-2
What Is Processor-in-the-Loop Simulation?	39-2
Why Use PIL Simulation?	39-2
Problems That PIL Simulation Can Detect	39-3
How PIL Simulation Works	39-3
Choosing a PIL Approach	39-5
Comparing Methods for Applying PIL Simulations	39-5
Modeling Scenarios with the PIL Block	39-5

Programming PIL Support for Third-Party Tools and Target Hardware	39-8
PIL Code Interfaces	39-9
Setting Up PIL Simulations With the Embedded IDE	
Link Product PIL Block	39-10
Configuring a PIL Simulation	39-14
Configuring a Top-Model PIL Simulation	39-14
Configuring PIL Mode in the Model Block	39-16
Code Coverage for a PIL Simulation	39-16
Creating a Connectivity Configuration for a Target ...	39-18
What Is a PIL Connectivity Configuration?	39-18
Overview of the Target Connectivity API	39-19
Creating a Connectivity API Implementation	39-22
Registering a Connectivity API Implementation	39-22
Demos of the Target Connectivity API	39-22
SIL and PIL Simulation Support and Limitations	39-24
SIL and PIL Simulation Support	39-25
Code Source Support	39-26
Block Support	39-27
Configuration Parameters Support	39-29
I/O Support	39-33
Hardware Implementation Support	39-42
Other Feature Support	39-46
PIL Block Limitations	39-46

Verifying Numerical Equivalence of Results with Code Generation Verification API

40

Verifying Numerical Equivalence with Code Generation Verification	40-2
Code Generation Verification API Overview	40-2
Verifying Numerical Equivalence with CGV Workflow ...	40-2

Example of Verifying Numerical Equivalence Between Two Modes of Execution of a Model	40-3
Example of Plotting Output Signals	40-10

Examples

A

Code Generation	A-2
Custom Storage Classes	A-2
Memory Sections	A-2
Advanced Code Generation	A-3
Target Function Libraries	A-3
Data Structures, Code Modules, and Program Execution	A-4
Verifying Generated Code	A-4

Index

Introduction to the Real-Time Workshop Embedded Coder Product

The Real-Time Workshop® Embedded Coder™ product *extends* the Real-Time Workshop® product with features that are important for embedded software development. Using the Real-Time Workshop Embedded Coder add-on product, you gain access to all aspects of Real-Time Workshop technology and can generate code that has the clarity and efficiency of professional handwritten code. For example, you can

- Generate code that is compact and fast, which is essential for real-time simulators, on-target rapid prototyping boards, microprocessors used in mass production, and embedded systems
- Customize the appearance of the generated code
- Optimize the generated code for a specific target environment
- Integrate existing (legacy) applications, functions, and data
- Enable tracing, reporting, and testing options that facilitate code verification activities

For detailed information on how the Real-Time Workshop Embedded Coder product fits into the complete Real-Time Workshop technology picture, see “Getting Started with Real-Time Workshop Technology” in the Real-Time Workshop documentation. That topic positions the Real-Time Workshop Embedded Coder in terms of what you can accomplish with it, how it can fit into your development process, how you might apply it to the V-model for system development, and how you might apply it to relevant use cases.

Because Real-Time Workshop Embedded Coder extends Real-Time Workshop for code generation, to use the Real-Time Workshop Embedded Coder product effectively, you should be familiar with information in parts of the Real-Time Workshop documentation that align with corresponding parts in the Real-Time Workshop Embedded Coder documentation.

- “Introduction to Real-Time Workshop Technology”
- “Developing Models for Code Generation”
- “Defining Data Representation and Storage for Code Generation”
- “Preparing Models for Code Generation”
- “Generating Code and Building Executables”
- “Integrating External Code With Generated C and C++ Code”
- “Setting Up Generated Code To Interface With Components in the Run-Time Environment”
- “Verifying Generated Code Applications”

Developing Models for Code Generation

- Chapter 2, “Setting Up Your Modeling Environment”
- Chapter 3, “Architecture Considerations”
- Chapter 4, “Scheduling Considerations”
- Chapter 5, “Developing Model Patterns that Generate Specific C Constructs”

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 the Real-Time Workshop Embedded Coder product to meet goals for specific domains.

Goals	Related Product Information	Demos
Generate a software design description	Simulink® Report Generator™ Simulink Report Generator documentation	rtwdemo_codegenrpt
Trace model requirements to generated code	Simulink® Verification and Validation™ “Including Requirements Information with Generated Code” in the Simulink Verification and Validation documentation	rtwdemo_requirements

Goals	Related Product Information	Demos
Implement application on fixed-point processors	Simulink® Fixed Point™ “Data Types and Scaling” and “Code Generation” in the Simulink Fixed Point documentation	rtwdemo_fixpt1 rtwdemo_fuelsys_fxp_publish rtwdemo_dspanc_fixpt
Use an integrated development environment (IDE) to integrate an application on a target processor automatically	Embedded IDE Link™ “Embedded IDE Link” documentation Target Support Package™ “Target Support Package” documentation	See help for Embedded IDE Link and Target Support Package products

Architecture Considerations

- “Generating Code Variants for Variant Models” on page 3-2
- “Creating and Using Host-Based Shared Libraries” on page 3-9

Generating Code Variants for Variant Models

In this section...
“Prerequisites” on page 3-2
“Why Generate Code Variants?” on page 3-2
“How to Generate Code Variations for Model Variants” on page 3-3
“Reviewing Code Variants in the Code Generation Report” on page 3-5
“Example of Variants in the Generated Code” on page 3-6
“Demo for Code Variants Using Model Blocks” on page 3-8

Prerequisites

The Real-Time Workshop Embedded Coder software generates code variants from a Simulink® model containing model reference variants. To learn how to create a model containing model reference variants, see “Using Model Reference Variants” in the Simulink documentation. The Real-Time Workshop Embedded Coder software generates code for all variants of a referenced model. In the generated code, variants are preprocessor conditional directives that determine the active variant and the sections of code to execute.

Note You cannot generate code variants for models with continuous states.

Why Generate Code Variants?

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

- Reuse generated code from a set of application models that share functionality with minor variations.
- Share generated code with a third party that activates one of the variants in the code.
- Validate all of the supported variants for a model and then choose to activate one variant for a particular application, without regenerating and revalidating the code.

How to Generate Code Variations for Model Variants

Defining Variant Control Variables and Variant Objects for Generating Code

To learn about variant control variables and variant objects see Implementing Model Reference Variants in the Simulink documentation. Variant control variables used for code generation have different requirements than variant control variables used for simulation. Perform the following steps to define variant control variables for generating code.

- 1 Open the Model Explorer and click the **Base Workspace**.
- 2 A variant control variable can be a `Simulink.Parameter` object or a `mpt.Parameter` object. In the Model Explorer, select **Add** and choose either **Simulink Parameter** or **MPT Parameter**. Specify a name for the new parameter.
- 3 On the `Simulink.Parameter` or `mpt.Parameter` property dialog box, specify the **Storage class** parameter by choosing one of the following:
 - `ImportedDefine(Custom)` custom storage class.
 - `CompilerFlag(Custom)` custom storage class.
 - A user-defined storage class created using the Custom Storage Class Designer. Your storage class must have the **Data initialization** parameter set to `Macro` and the **Data scope** parameter set to `Imported`. See “Using the Custom Storage Class Designer” on page 7-12 for more information.

Note If the storage class is `CompilerFlag(Custom)` then the makefile options (for example, `OPTOPTS`) must specify the macro and its value.

- 4 If the storage class is either `ImportedDefine(Custom)` or a user-defined custom storage class, specify the **Header File** parameter as an external header file in the Custom Attributes section of the `Simulink.Parameter` property dialog box.

- 5 Supply the values of the variant control variables in the external header file. The generated code refers to a variant control variable as a user-defined macro. The generated code does not contain the value of the macro. The value of the variant control variable determines the active variant in the compiled code. You can use a MATLAB
- 6 Follow the instructions for “Implementing Variant Objects” for model reference variants to implement variant objects for code generation. Ensure that only one variant object is active in the generated code by implementing the condition expressions of the variant objects such that only one evaluates to true. The generated code includes a test of the variant objects to determine that there is only one active variant. If this test fails, your code might not compile.

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

Generating Preprocessor Conditional Directives

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

- 1 On the **Optimization** pane of the Configuration Parameters dialog box, select **Inline parameters**.
- 2 On the **Real-Time Workshop** pane of the Configuration Parameter dialog box, clear “Ignore custom storage classes”. In order to generate preprocessor conditionals, you must use custom storage classes.
- 3 On the **Interface** pane of the Configuration Parameter dialog box, select the **Use Local Settings** option of the **Generate preprocessor conditionals** parameter. This parameter is a global setting for the parent model. This setting enables the **Generate preprocessor conditionals** parameter located in the Model block parameters dialog box. See “Generate preprocessor conditionals” for more information.

- 4 Open the Model block parameters dialog box for a variant model block. Select the **Generate preprocessor conditionals** parameter.
- 5 In the Model block parameters dialog box, clear the **Override variant conditions and use following variant** parameter.
- 6 Build your model.

Reviewing Code Variants in the Code Generation Report

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

Code Variants Report for rtwdemo_preprocessor

Table of Contents

- [Variant Objects](#)
- [Model Blocks that have Variants](#)

Variant Objects

[-]

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

Model Blocks that have Variants

[-]

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

Example of Variants in the Generated Code

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

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

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

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

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

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

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

- A check to ensure that exactly one variant is active at a time:

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

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

```
/* ModelReference: '<Root>/Left Controller' */
#if LINEAR

mr_rtwdemo_linl(&rtb_Add, &rtb_LeftController_merge_1,
                &(rtwdemo_preprocessor_DWork.LeftController_1_DWORK1.rtdw));

#endif                                /* LINEAR */
```

```
/* ModelReference: '<Root>/Left Controller' */
#if NONLINEAR

mr_rtwdemo_nlinl(&rtb_Add, &rtb_LeftController_merge_1,
                &(rtwdemo_preprocessor_DWork.LeftController_2_DWORK1.rtdw));

#endif                                     /* NONLINEAR */
```

and

```
/* ModelReference: '<Root>/Right Controller' */
#if LINEAR

mr_rtwdemo_linr(&rtb_Add1, &rtb_RightController_merge_1,
                &(rtwdemo_preprocessor_DWork.RightController_1_DWORK1.rtdw));

#endif                                     /* LINEAR */

/* ModelReference: '<Root>/Right Controller' */
#if NONLINEAR

mr_rtwdemo_nlinr(&rtb_Add1, &rtb_RightController_merge_1,
                 &(rtwdemo_preprocessor_DWork.RightController_2_DWORK1.rtdw));

#endif                                     /* NONLINEAR */
```

Demo for Code Variants Using Model Blocks

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

Creating and Using Host-Based Shared Libraries

In this section...

“Overview” on page 3-9

“Generating a Shared Library Version of Your Model Code” on page 3-10

“Creating Application Code to Load and Use Your Shared Library File” on page 3-11

“Host-Based Shared Library Limitations” on page 3-15

Overview

The Real-Time Workshop Embedded Coder product provides an ERT target, `ert_shrlib.tlc`, for generating a host-based shared library from your Simulink model. Selecting this target allows you to generate a shared library version of your model code that is appropriate for your host platform, either a Windows® dynamic link library (`.dll`) file or a UNIX® shared object (`.so`) file. This feature can be used to package your source code securely for easy distribution and shared use. The generated `.dll` or `.so` file is shareable among different applications and upgradeable without having to recompile the applications that use it.

Code generation for the `ert_shrlib.tlc` target exports

- Variables and signals of type `ExportedGlobal` as data
- Real-time model structure (`model_M`) as data
- Functions essential to executing your model code

To view a list of symbols contained in a generated shared library file, you can

- On Windows, use the Dependency Walker utility, downloadable from <http://www.dependencywalker.com>
- On UNIX, use `nm -D model.so`

To generate and use a host-based shared library, you

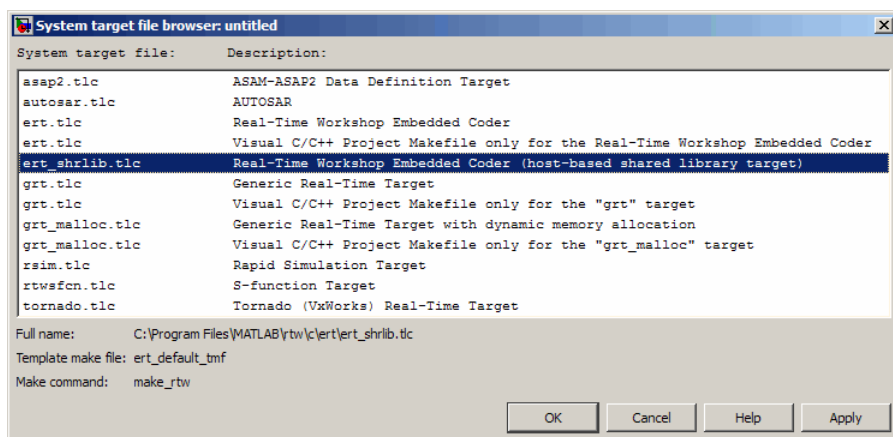
- 1 Generate a shared library version of your model code

- 2 Create application code to load and use your shared library file

Generating a Shared Library Version of Your Model Code

This section summarizes the steps needed to generate a shared library version of your model code.

- 1 To configure your model code for shared use by applications, open your model and select the `ert_shrlib.tlc` target on the **Real-Time Workshop** pane of the Configuration Parameters dialog box. Click **OK**.



Selecting the `ert_shrlib.tlc` target causes the build process to generate a shared library version of your model code into your current working folder. The selection does not change the code that is generated for your model.

- 2 Build the model.
- 3 After the build completes, you can examine the generated code in the model subfolder, and the `.dll` file or `.so` file that has been generated into your current folder.

Creating Application Code to Load and Use Your Shared Library File

To illustrate how application code can load an ERT shared library file and access its functions and data, The MathWorks™ provides the demo model `rtwdemo_shrlib`. Clicking the blue button in the demo model runs a script that:

- 1 Builds a shared library file from the model (for example, `rtwdemo_shrlib_win32.dll` on 32-bit Windows)
- 2 Compiles and links an example application, `rtwdemo_shrlib_app`, that will load and use the shared library file
- 3 Executes the example application

Note It is recommended that you change directory to a new or empty folder before running the `rtwdemo_shrlib` script.

The demo model uses the following example application files, which are located in `matlabroot/toolbox/rtw/rtwdemos/shrlib_demo`.

File	Description
<code>rtwdemo_shrlib_app.h</code>	Example application header file
<code>rtwdemo_shrlib_app.c</code>	Example application that loads and uses the shared library file generated for the demo model
<code>run_rtwdemo_shrlib_app.m</code>	Script to compile, link, and execute the example application

You can view each of these files by clicking white buttons in the demo model window. Additionally, running the script places the relevant source and generated code files in your current folder. The files can be used as templates for writing application code for your own ERT shared library files.

The following sections present key excerpts of the example application files.

Example Application Header File

The example application header file `rtwdemo_shrlib_app.h` contains type declarations for the demo model's external input and output.

```
#ifndef _APP_MAIN_HEADER_
#define _APP_MAIN_HEADER_

typedef struct {
    int32_T Input;
} ExternalInputs_rtwdemo_shrlib;

typedef struct {
    int32_T Output;
} ExternalOutputs_rtwdemo_shrlib;

#endif /* _APP_MAIN_HEADER_ */
```

Example Application C Code

The example application `rtwdemo_shrlib_app.c` includes the following code for dynamically loading the shared library file. Notice that, depending on platform, the code invokes Windows or UNIX library commands.

```
#if (defined(_WIN32)||defined(_WIN64)) /* WINDOWS */
#include <windows.h>
#define GETSYMBOLADDR GetProcAddress
#define LOADLIB LoadLibrary
#define CLOSELIB FreeLibrary

#else /* UNIX */
#include <dlfcn.h>
#define GETSYMBOLADDR dlsym
#define LOADLIB dlopen
#define CLOSELIB dlclose

#endif

int main()
{
    void* handleLib;
```

```

...
#if defined(_WIN64)
    handleLib = LOADLIB("./rtwdemo_shrlib_win64.dll");
#else
#if defined(_WIN32)
    handleLib = LOADLIB("./rtwdemo_shrlib_win32.dll");
#else /* UNIX */
    handleLib = LOADLIB("./rtwdemo_shrlib.so", RTLD_LAZY);
#endif
#endif
#endif
...
    return(CLOSELIB(handleLib));
}

```

The following code excerpt shows how the C application accesses the demo model's exported data and functions. Notice the hooks for adding user-defined initialization, step, and termination code.

```

    int32_T i;
...
    void (*mdl_initialize)(boolean_T);
    void (*mdl_step)(void);
    void (*mdl_terminate)(void);

    ExternalInputs_rtwdemo_shrlib (*mdl_Uptr);
    ExternalOutputs_rtwdemo_shrlib (*mdl_Yptr);

    uint8_T (*sum_outptr);
...
#if (defined(LCCDLL)||defined(BORLANDCDLL))
/* Exported symbols contain leading underscores when DLL is linked with
   LCC or BORLANDC */
mdl_initialize =(void*)(boolean_T)GETSYMBOLADDR(handleLib ,
        "_rtwdemo_shrlib_initialize");
mdl_step      =(void*)(void)GETSYMBOLADDR(handleLib ,
        "_rtwdemo_shrlib_step");
mdl_terminate =(void*)(void)GETSYMBOLADDR(handleLib ,
        "_rtwdemo_shrlib_terminate");
mdl_Uptr     =(ExternalInputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
        "_rtwdemo_shrlib_U");

```

```
mdl_Yptr      =(ExternalOutputs_rtwdemo_shrplib*)GETSYMBOLADDR(handleLib ,
    "_rtwdemo_shrplib_Y");
sum_outptr    =(uint8_T*)GETSYMBOLADDR(handleLib , "_sum_out");
#else
mdl_initialize =(void*)(boolean_T)GETSYMBOLADDR(handleLib ,
    "rtwdemo_shrplib_initialize");
mdl_step      =(void*)(void)GETSYMBOLADDR(handleLib ,
    "rtwdemo_shrplib_step");
mdl_terminate =(void*)(void)GETSYMBOLADDR(handleLib ,
    "rtwdemo_shrplib_terminate");
mdl_Uptr      =(ExternalInputs_rtwdemo_shrplib*)GETSYMBOLADDR(handleLib ,
    "rtwdemo_shrplib_U");
mdl_Yptr      =(ExternalOutputs_rtwdemo_shrplib*)GETSYMBOLADDR(handleLib ,
    "rtwdemo_shrplib_Y");
sum_outptr    =(uint8_T*)GETSYMBOLADDR(handleLib , "sum_out");
#endif

if ((mdl_initialize && mdl_step && mdl_terminate && mdl_Uptr && mdl_Yptr &&
    sum_outptr)) {
    /* === user application initialization function === */
    mdl_initialize(1);
    /* insert other user defined application initialization code here */

    /* === user application step function === */
    for(i=0;i<=12;i++){
        mdl_Uptr->Input = i;
        mdl_step();
        printf("Counter out(sum_out): %d\tAmplifier in(Input): %d\tout(Output): %d\n",
            *sum_outptr, i, mdl_Yptr->Output);
        /* insert other user defined application step function code here */
    }

    /* === user application terminate function === */
    mdl_terminate();
    /* insert other user defined application termination code here */
}
else {
    printf("Cannot locate the specified reference(s) in the shared library.\n");
    return(-1);
}
```

Example Application Script

The application script `run_rtwdemo_shrlib_app.m` loads and rebuilds the demo model, and then compiles, links, and executes the demo model's shared library target file. You can view the script source file by opening `rtwdemo_shrlib` and clicking the appropriate white button. The script constructs platform-dependent command strings for compilation, linking, and execution that may apply to your development environment. To run the script, click the blue button.

Host-Based Shared Library Limitations

The following limitations apply to using ERT host-based shared libraries:

- Code generation for the `ert_shrlib.tlc` target exports only the following as data:
 - Variables and signals of type `ExportedGlobal`
 - Real-time model structure (`model_M`)
- Code generation for the `ert_shrlib.tlc` target supports the C language only (not C++). When you select the `ert_shrlib.tlc` target, language selection is greyed out on the **Real-Time Workshop** pane of the Configuration Parameters dialog box.
- On Windows systems, the `ert_shrlib` target by default does not generate or retain the `.lib` file for implicit linking (explicit linking is preferred for portability).

You can change the default behavior and retain the `.lib` file by modifying the corresponding template makefile (TMF). If you do this, be aware that the generated `model.h` file will need a small modification to be used together with the generated `ert_main.c` for implicit linking. For example, if you are using Visual C++®, you will need to declare `__declspec(dllimport)` in front of all data to be imported implicitly from the shared library file.

- To reconstruct a model simulation using a generated host-based shared library, the application author must maintain the timing between system and shared library function calls in the original application. The timing needs to be consistent to ensure correct simulation and integration results.

Scheduling Considerations

- “Using Discrete and Continuous Time” on page 4-2
- “Optimizing Task Scheduling for Multirate Multitasking Models on RTOS Targets” on page 4-4

Using Discrete and Continuous Time

In this section...
“Generating Code for Discrete and Continuous Time Blocks” on page 4-2
“Generating Code that Supports Continuous Solvers” on page 4-2
“Generating Code that Honors a Stop Time” on page 4-3

Generating Code for Discrete and Continuous Time Blocks

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

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

```
showblockdatatypetable
```

Generating Code that Supports Continuous Solvers

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

Note Custom targets must be modified to support continuous time. The required modifications are described in “Custom Targets” in the Real-Time Workshop documentation.

Generating Code that Honors a Stop Time

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

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

Otherwise, the executable runs indefinitely.

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

Optimizing Task Scheduling for Multirate Multitasking Models on RTOS Targets

In this section...

“Overview” on page 4-4

“Using `rtmStepTask`” on page 4-5

“Task Scheduling Code for Multirate Multitasking Model on Wind River Systems VxWorks Target” on page 4-5

“Suppressing Redundant Scheduling Calls” on page 4-6

Overview

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

To understand the optimization that is available for an RTOS target, consider how the ERT target schedules tasks for bareboard targets (where no RTOS is present). The ERT target maintains *scheduling counters* and *event flags* for each subrate task. The scheduling counters are implemented within the real-time model (rtM) data structure as arrays, indexed on task identifier (`tid`).

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

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

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

For bareboard targets that cannot rely on an external RTOS, the event flags are mandatory to allow overlapping task preemption. However, an RTOS target uses the operating system itself to manage overlapping task preemption, making the maintenance of the event flags redundant.

Using `rtmStepTask`

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

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

The arguments are:

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

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

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

Task Scheduling Code for Multirate Multitasking Model on Wind River Systems VxWorks Target

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

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

Suppressing Redundant Scheduling Calls

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

```
%assign SuppressSetEventsForThisBaseRateFcn = 1
```

Developing Model Patterns that Generate Specific C Constructs

- “About Modeling Patterns” on page 5-2
- “Standard Methods to Prepare a Model for Code Generation” on page 5-3
- “Types, Operators, and Expressions” on page 5-8
- “Control Flow” on page 5-25
- “Functions” on page 5-64

About Modeling Patterns

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

Model examples have the following naming conventions:

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

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

Standard Methods to Prepare a Model for Code Generation

In this section...

“Configuring a Signal” on page 5-3

“Configuring Input and Output Ports” on page 5-4

“Initializing States” on page 5-4

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

“Setting Up an Example Model With a Stateflow Chart” on page 5-5

“Setting Up an Example Model With an Embedded MATLAB Block” on page 5-7

Configuring a Signal

- 1 Create a model in Simulink. See “Creating a Model” in the Simulink documentation.
- 2 Right-click a signal line. Select **Signal Properties**. A Signal Properties dialog box opens. See “Signal Properties Dialog Box” for more information.
- 3 Enter a signal name for the **Signal name** parameter.
- 4 On the same Signal Properties dialog box, select the **Real-Time Workshop** tab. Use the drop down menu for the **Storage class** parameter to specify a storage class. Examples in this chapter use Exported Global.

Note Alternatively, on the Signal Properties dialog box, select **Signal name must resolve to Simulink signal object**. Then create a signal data object in the base workspace with the same name as the signal. See “Creating Simulink and mpt Data Objects” on page 11-4 for more information on creating data objects in the base workspace. (Examples use `mpt.Signal` and specify the **Storage class** as `ExportedGlobal`.)

Configuring Input and Output Ports

1 In your model,

Double-click an Inport or Outport block. A Block Parameters dialog box opens.

2 Select the **Signal Attributes** tab.

3 Specify the **Port dimensions** and **Data type**. Examples leave the default value for **Port dimensions** as 1 (for inherited) and **Data type** as Inherit: auto.

Initializing States

1 Double-click a block.

2 In the Block Parameters dialog box, select the **Main** tab.

3 Specify the **Initial conditions** and **Sample time**. See “Working with Sample Times”.

4 Select the **State Attributes** pane. Specify the state name. See “Block State Storage and Interfacing Considerations”

5 You can also use the Data Object Wizard for creating data objects. A part of this process initializes states. See “Creating Simulink Data Objects with Data Object Wizard” on page 11-5.

Setting Up Configuration Parameters for Code Generation

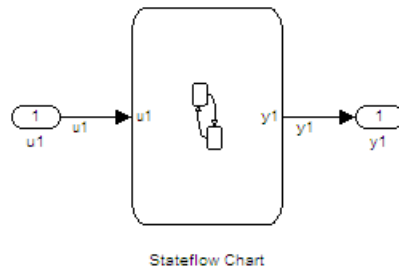
1 Open the Configuration Parameter dialog box by selecting **Simulation > Configuration parameters**. You can also use the keyboard shortcut Ctrl+E.

2 Open the **Solver** pane and select

- **Solver type:** Fixed-Step
- **Solver:** Discrete (no continuous states)

- 3** Open the **Optimization** pane, and select the **Inline parameters** parameter.
- 4** Open the **Real-Time Workshop** pane, and specify `ert.tlc` as the **System Target File**.
- 5** Clear **Generate makefile**.
- 6** Select **Generate code only**.
- 7** Enable the HTML report generation by opening the **Real-Time Workshop** > **Report** pane and selecting **Create code generation report**, **Launch report automatically**, and **Code-to-model**. Enabling the HTML report generation is optional.
- 8** Click **Apply** and then **OK** to exit.

Setting Up an Example Model With a Stateflow Chart



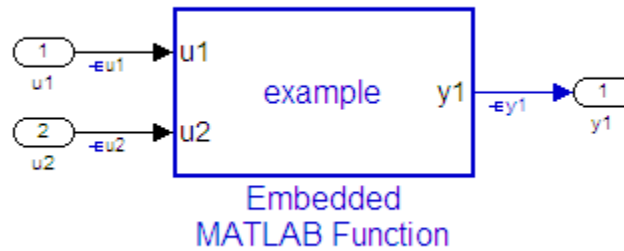
Follow this general procedure to create a simple model containing a Stateflow chart.

- 1** From the **Stateflow** > **Stateflow Chart** library, add a Stateflow chart to your model .
- 2** Add Inport blocks and Outport blocks according to the example model.
- 3** Open the **Stateflow Editor** by performing on teh the following:

- Double-click the Stateflow chart.
 - Select **Tools > Explore**.
 - Press **Ctrl+R**.
- 4** Select **Add > Data > Input from Simulink** to add the inputs to the chart. A Data dialog box opens for each input.
 - 5** Specify the **Name** (u1, u2, ...) and the **Type** (Inherit: same as Simulink, unless specified differently in the example) for each input and click **OK**.

Click **Apply** and close each dialog box.
 - 6** Select **Add > Data > Output from Simulink** to add the outputs to the chart. A Data dialog opens for each output.
 - 7** Specify the **Name** (y1, y2, ...) and **Type** (Inherit: same as Simulink, unless specified differently in the example) for each output and click **OK**.
 - 8** Click **Apply** and close each dialog box.
 - 9** In the **Stateflow Editor**, create the Stateflow diagram specific to the example.
 - 10** The inputs and outputs appear on the chart in your model.
 - 11** Connect the Inport and Outport blocks to the Stateflow Chart.
 - 12** Configure the input and output signals; see “Configuring a Signal” on page 5-3.

Setting Up an Example Model With an Embedded MATLAB Block



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

Types, Operators, and Expressions

In this section...
“Data Declaration” on page 5-8
“Data Type Conversion” on page 5-11
“Type Qualifiers” on page 5-15
“Relational and Logical Operators” on page 5-17
“Bitwise Operations” on page 5-21

Data Declaration

C Construct

```
int32 p1 = 3;
```

Declare a Variable for a Block Parameter Using a Data Object

You can specify certain block parameters as a variable. If you define the variable as a data object, the variable is global. Where the variable is declared in the generated code depends on the custom storage class that you choose (and whether you select Optimization parameter **Inline Parameters**). If you choose **Inline Parameters**, then the data object name is used in the generated code. If you did not choose **Inline Parameters**, the generated code creates a global structure that stores all of the parameters. For more information on how to create a data object, see Defining Data Representation and Storage for Code Generation on page 1.

Block	Parameter
Constant	Value
Gain	Value
For Iterator	Iteration Limit

There are several methods for configuring data objects:

- For a model with many parameters, use the Data Object Wizard, which analyzes your model and finds all of the unresolved data objects and data types. You can then create the data objects in the Data Object Wizard. The procedure for using the Data Object Wizard for a parameter is similar to the procedure for a signal. For an example, see “Declare a Variable for a Signal using a Data Object” on page 5-10.
- To add, delete, edit, and configure data objects, use the base workspace in the Model Explorer.
- To create and configure data objects, use the MATLAB® command line.

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

```
int Kp = 3;
```

- 1** Create a model containing a Constant block and a Gain block.
- 2** Press **Ctrl+E** to open the Configuration Parameters dialog box.
- 3** On the **Optimization** pane of the Configuration Parameter dialog box, select **Inline parameters**.
- 4** Click **Apply** and **OK**. The Configuration Parameter dialog box closes.
- 5** In your model, double-click the Constant block. The Block Parameters dialog box opens.
- 6** In the **Value** field, enter a variable name. In this example, the variable name is p1.
- 7** In your model, double-click the Gain block. The Block Parameters dialog box opens.
- 8** In the **Value** field, enter a variable name. In this example, the variable name is p2.
- 9** Press **Ctrl+H** to open the Model Explorer. On the Model Hierarchy pane, select the base workspace.

- 10** To add two MPT parameter objects, in the menu bar, select **Add > MPT Parameter** in the menu bar twice. On the **Contents of: Base Workspace** pane, you see the parameters.
- 11** Double-click each `mpt.Parameter` object and change their names to `p1` and `p2`.
- 12** Click the `p1` parameter. The data object parameters are displayed in the right pane of the Model Explorer.
- 13** In the **Value** field, enter 3 for `p1`. For the **Data type**, select `int32`. Because you chose an MPT parameter, the **Storage Class** is already set to `Global(Custom)`.
- 14** In the **Value** field, enter 5 for `p2`. For the **Data type**, select `int32`.
- 15** Press **Ctrl+B** to generate code.

In the `model.c` file you see:

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

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

C Construct

```
int p1 = 3;
```

Declare a Variable for a Signal using a Data Object

- 1** Create a model and label the signals.
- 2** In the model tool bar, click **Tools > Data Object Wizard** to open the **Data Object Wizard**. If you are not familiar with creating Simulink Data Objects using the wizard, refer to “Data Object Wizard” .

- 3** Click **Find**. The list of unresolved parameters and objects populates the Data Object Wizard. You can do mass edits for identical data objects.
- 4** Select the signals individually or select all signals by clicking **Check All**.
- 5** From the parameter **Choose package for selected data objects** drop-down list, select the `mpt` package. Click **Apply Package**. When you open the Model Explorer the data objects appear in the base workspace.
- 6** In the base workspace, click the `p1` data object. The data object parameters appear in the right pane of the Model Explorer.
- 7** From the **Data type** drop-down list, select `int16`.
- 8** You can also specify the storage class. The data object is an `mpt.Parameter` object, therefore the Storage Class is automatically set to `Global (Custom)`.

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

Data Type Conversion

C Construct

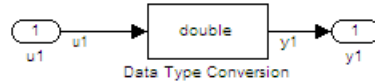
```
y1 = (double)u1;
```

Modeling Patterns

- “Modeling Pattern for Data Type Conversion — Simulink Block” on page 5-12
- “Modeling Pattern for Data Type Conversion — Stateflow Chart” on page 5-13
- “Modeling Pattern for Data Type Conversion — Embedded MATLAB Function” on page 5-14

Modeling Pattern for Data Type Conversion – Simulink Block

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



Data_Type_SL

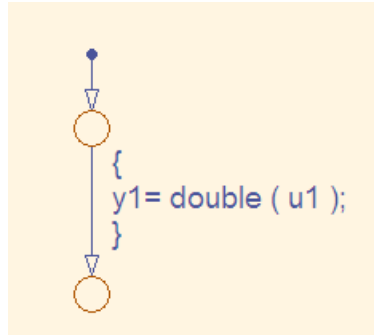
- 1 From the **Commonly Used Blocks** library, drag a Data Type Conversion block into your model and connect to the Inport and Outport blocks.
- 2 Double-click on the Data Type Conversion block to open the Block Parameters dialog box. and in the Block Parameters dialog box select the .
- 3 Select the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

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

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

The Real-Time Workshop Embedded Coder type definition for double is `real_T`.

Modeling Pattern for Data Type Conversion – Stateflow Chart



Stateflow Chart Type Conversion

Procedure.

- 1 Follow the steps for “Setting Up an Example Model With a Stateflow Chart” on page 5-5 . This example contains one Inport block and one Outport block.
- 2 Name the example model `Data_Type_SF`.
- 3 Double-click the Inport block and select the **Signal Attributes** tab. Specify the **Data Type** as `int32` from the drop down menu.
- 4 Double-click the Outport block and select the **Signal Attributes** tab. Specify the **Data Type** as `Inherit: auto` from the drop down menu.
- 5 In the **Stateflow Editor**, specify the **Data Type** for `y1` as `Boolean`
- 6 Press **Ctrl+B** to build the model and generate code.

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

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

Modeling Pattern for Data Type Conversion – Embedded MATLAB Function

Procedure.

- 1 Follow the steps for “Setting Up an Example Model With an Embedded MATLAB Block” on page 5-7 . This example model contains one Inport block and one Outport block.
- 2 Name the model Data_Type_EML.
- 3 In the Embedded MATLAB editor enter the function, as follows:

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

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

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

```
real32_T u1;
real_T y1;

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

Other Type Conversions in Modeling

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

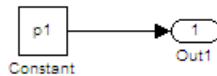
Type Qualifiers

Modeling Patterns for Type Qualifiers

- “Using a Tunable Parameter in the Base Workspace” on page 5-15
- “Using a Data Object of the Const Custom Storage Class” on page 5-16

Using a Tunable Parameter in the Base Workspace

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



Type_Qual

Procedure.

- 1 Create a model containing a Constant block and an Outport block.
- 2 Double-click the Constant block. In the **Constant value** field, enter the parameter name p1 .
- 3 In the base workspace, create a MATLAB variable for p1 and specify its **Value** as 9.8 and its **Data type** as double.
- 4 Press **Ctrl+E** to open the Configuration Parameters dialog box and select the **Optimization** pane.
- 5 Select the **Inline parameters** parameter, which activates the **Configure** button.
- 6 Click the **Configure** button to open the Model Parameter Configuration dialog box.

- 7 To declare a tunable parameter, from the **Source list**, select the variable p1.
- 8 Click the **Add to table** button to add p1 to the **Global (tunable) parameters** section.
- 9 Click the **Storage Class** and select Exported Global.
- 10 Click the **Storage Type Qualifier** arrow and select const.
- 11 Click **Apply** to save all of the changes.
- 12 Press **Ctrl+B** to build the model and generate code.

Results. The generated code appears in `Type_Qual.c` as follows:

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

Using a Data Object of the Const Custom Storage Class

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

Procedure.

- 1 Press **Ctrl+H** to open the Model Explorer. On the Model Hierarchy pane, select the base workspace.
- 2 In the menu bar, select **Add > MPT Parameter** to add an MPT parameter object. The parameter is displayed in the **Contents of: Base Workspace** pane.
- 3 Double-click the `mpt.Parameter` object and change the **Name** to p1.
- 4 Click the p1 parameter which displays the data object parameters on the right pane of the Model Explorer.

- 5** In the **Value** field, enter **9.8** for **p1**. Specify the **Data type** as **auto** for 64-bit double.
- 6** You can use the different type qualifiers by selecting a custom storage class from the **Storage class** list. For this example, select **ConstVolatile (custom)**.
- 7** In the Configuration Parameters dialog box, on the **Optimization** pane, select the **Inline parameters**.
- 8** Press **Ctrl+B** to build the model and generate code.

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

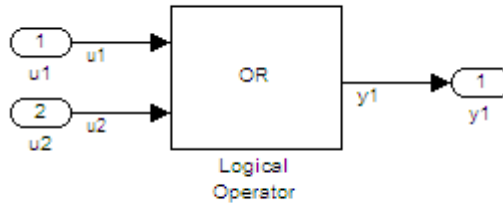
```
const volatile real_T p1 = 9.8;
```

Relational and Logical Operators

Modeling Patterns for Relational and Logical Operators

- “Modeling Pattern for Relational or Logical Operators — Simulink Blocks” on page 5-18
- “Modeling Pattern for Relational and Logical Operators — Stateflow Chart” on page 5-19
- “Modeling Pattern for Relational and Logical Operators — Embedded MATLAB” on page 5-20

Modeling Pattern for Relational or Logical Operators – Simulink Blocks



Logical_SL

Procedure.

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

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

Results. Code implementing the logical operator OR is in the Logical_SL_step function in Logical_SL.c:

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

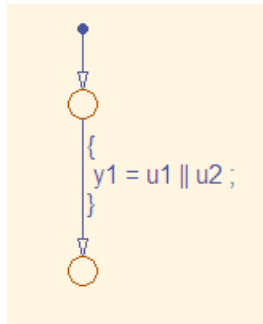
```

boolean_T u2;                                /* '<Root>/u2' */
boolean_T y1;                                /* '<Root>/Logical Operator'*/

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

```

Modeling Pattern for Relational and Logical Operators –Stateflow Chart



Logical SF/Logical Operator Stateflow® Chart

Procedure.

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

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

Results. Code implementing the logical operator OR is in the Logical_SF_step function in Logical_SF.c:

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

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

Modeling Pattern for Relational and Logical Operators – Embedded MATLAB

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

Procedure.

- 1** Follow the steps for “Setting Up an Example Model With an Embedded MATLAB Block” on page 5-7 . This example model contains two Inport blocks and one Outport block.
- 2** Name the example model Rel_Operator_EML.
- 3** In the Embedded MATLAB editor enter the function, as follows:

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

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

Results. Code implementing the relational operator '>' is in the Rel_Operator_EML_step function in Rel_Operator_EML.c:

```
real_T u1;              /* '<Root>/u1' */
```



```

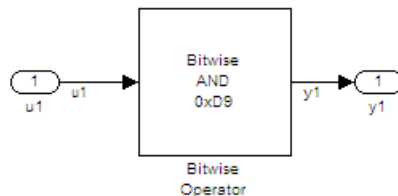
real_T u2;                               /* '<Root>/u2' */
boolean_T y;                              /* '<Root>/Embedded MATLAB Function' */

void Rel_Operator_EML_step(void)
{
    y = (u1 > u2);
}

```

Bitwise Operations

Simulink Bitwise-Operator Block



Bit_Logic_SL

Procedure.

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

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

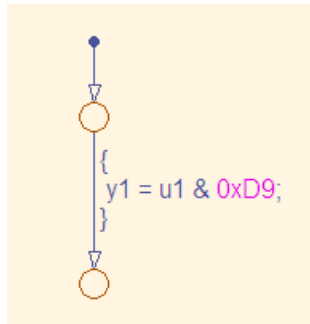
- 5 In the **Bit Mask** field, enter a decimal number. Use `bin2dec` or `hex2dec` to convert from binary or hexadecimal. In this example, enter `hex2dec('D9')`.
- 6 Name the blocks, as shown in, model `Bit_Logic_SL`.
- 7 Connect the blocks and name the signals, as shown in, model `Bit_Logic_SL`.
- 8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

Results. Code implementing the logical operator OR is in the `Bit_Logic_SL_step` function in `Bit_Logic_SL.c`:

```
uint8_T u1;
uint8_T y1;

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

Stateflow Chart



Bit_Logic_SF/Bit_Logic Stateflow Chart

Procedure.

- 1 Follow the steps for “Setting Up an Example Model With a Stateflow Chart” on page 5-5. This example contains one Inport block and one Outport block.

- 2** Name the example model `Bit_Logic_SF`.
- 3** Select **Tools > Explore** to open the Model Explorer by .
- 4** In the Model Explorer, on the right pane, select **Enable C-bit operations**.
- 5** In the **Stateflow Editor**, create the Stateflow diagram, `Bit_Logic_SF/Bit_Logic Stateflow Chart`.
- 6** Press **Ctrl+B** to build the model and generate code.

Results. Code implementing the logical operator OR is in the `Bit_Logic_SF_step` function in `Bit_Logic_SF.c`:

```
uint8_T u1;
uint8_T y1;

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

Embedded MATLAB Block

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

Procedure.

- 1** Follow the steps for “Setting Up an Example Model With an Embedded MATLAB Block” on page 5-7. This example model contains two Inport blocks and one Output block.
- 2** Name your model `Bit_Logic_EML`.
- 3** In the Embedded MATLAB editor enter the function, as follows:

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

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

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

Results. Code implementing the bitwise operator OR is in the `Bit_Logic_EML_step` function in `Bit_Logic_EML.c`:

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

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

Control Flow

In this section...

“If-Else” on page 5-25

“Switch” on page 5-32

“For loop” on page 5-38

“While loop” on page 5-46

“Do While loop” on page 5-57

If-Else

C Construct

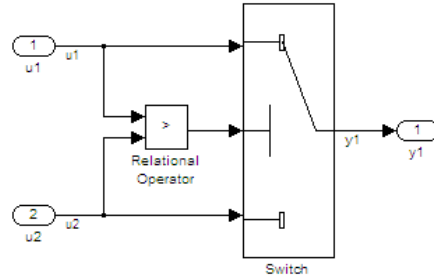
```
if (u1 > u2)
{
    y1 = u1;
}
else
{
    y1 = u2;
}
```

Modeling Patterns

- “Modeling Pattern for If-Else: Switch block” on page 5-26
- “Modeling Pattern for If-Else: Stateflow Chart” on page 5-28
- “Modeling Pattern for If-Else: Embedded MATLAB Block” on page 5-30

Modeling Pattern for If-Else: Switch block

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



Model If_Else_SL

Procedure.

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

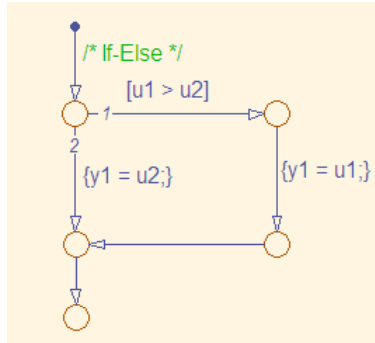
Results. Real-Time Workshop software generates the following If_Else_SL_step function in the file If_Else_SL.c:

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

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

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

Modeling Pattern for If-Else: Stateflow Chart



If-Else Stateflow Chart

Procedure.

- 1 Follow the steps for “Setting Up an Example Model With a Stateflow Chart” on page 5-5. This example model contains two Inport blocks and one Outport block.
- 2 Name your model `If_Else_SF`.
- 3 When configuring your Stateflow chart, select **Patterns > Add Decision > If-Else**. The Stateflow Pattern dialog opens. Fill in the fields as follows:

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

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

Results. Real-Time Workshop software generates the following `If_Else_SF_step` function in the file `If_Else_SF.c`:

```

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

```



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

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

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

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

Modeling Pattern for If-Else: Embedded MATLAB Block

Procedure.

- 1 Follow the steps for “Setting Up an Example Model With an Embedded MATLAB Block” on page 5-7. This example model contains two Inport blocks and one Output block.
- 2 Name your model `If_Else_EML`.
- 3 In the Embedded MATLAB editor enter the function, as follows:

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

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

Results. Real-Time Workshop software generates the following `If_Else_EML_step` function in the file `If_Else_EML.c`:

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

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

/* Model step function */
void If_Else_EML_step(void)
{
    /* Embedded MATLAB: '<Root>/Embedded MATLAB Function' incorporates:
    * Inport: '<Root>/u1'
    * Inport: '<Root>/u2'
    * Output: '<Root>/y1'
    */
    /* Embedded MATLAB Function 'Embedded MATLAB Function': '<S1>:1' */
    if (U.u1 > U.u2) {
        /* '<S1>:1:4' */
        /* '<S1>:1:5' */
    }
}
```

```
    Y.y1 = U.u1;
} else {
    /* '<S1>:1:6' */
    Y.y1 = U.u2;
}
}
```

Switch

C Construct

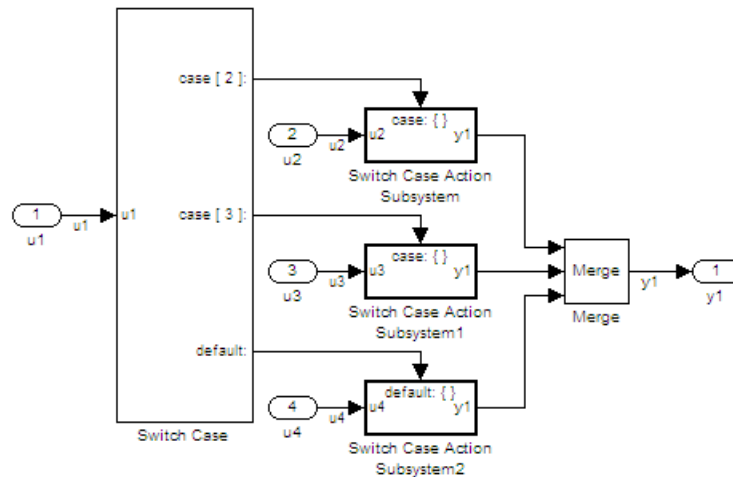
```
switch (u1)
{
  case 2:
    y1 = u2;
    break;
  case 3:
    y1 = u3;
    break;
  default:
    y1 = u4;
    break;
}
```

Modeling Patterns

- “Modeling Pattern for Switch: Switch Case block” on page 5-33
- “Modeling Pattern for Switch: Embedded MATLAB block” on page 5-36
- “Converting If-Elseif-Else to Switch statement” on page 5-37

Modeling Pattern for Switch: Switch Case block

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



Model Switch_SL

Procedure.

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

- 7 Drag a Merge block from the **Simulink > Signal Routing** library to merge the outputs.
- 8 The Switch Case block takes an integer input, therefore, the input signal `u1` is type cast to an `int32`.
- 9 Enter `Ctrl+B` to build the model and generate code.

Results. Real-Time Workshop software generates the following `Switch_SL_step` function in the file `Switch_SL.c`:

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

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

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

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

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

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

Modeling Pattern for Switch: Embedded MATLAB block

Procedure.

- 1 Follow the steps for “Setting Up an Example Model With an Embedded MATLAB Block” on page 5-7. This example model contains four Inport blocks and one Outport block.
- 2 Name your model `If_Else_EML`.
- 3 In the Embedded MATLAB editor enter the function, as follows:

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

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

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

Results. Real-Time Workshop software generates the following `Switch_EML_step` function in the file `Switch_EML.c`:

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

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

/* Model step function */
void Switch_EML_step(void)
{
    /* Embedded MATLAB: '<Root>/Embedded MATLAB Function' incorporates:
    * Inport: '<Root>/u1'
    * Inport: '<Root>/u2'
    * Inport: '<Root>/u3'
    */
}
```



```

* Inport: '<Root>/u4'
* Output: '<Root>/y1'
*/
/* Embedded MATLAB Function 'Embedded MATLAB Function': '<S1>:1' */
/* '<S1>:1:4' */
switch (U.u1) {
case 2:
/* '<S1>:1:6' */
Y.y1 = U.u2;
break;

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

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

```

Converting If-Elseif-Else to Switch statement

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

For loop

C Construct

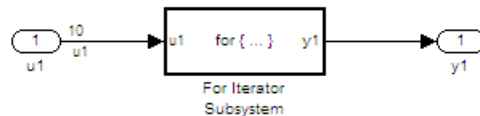
```
y1 = 0;
for(inx = 0; inx <10; inx++)
{
    y1 = u1[inx] + y1;
}
```

Modeling Patterns:

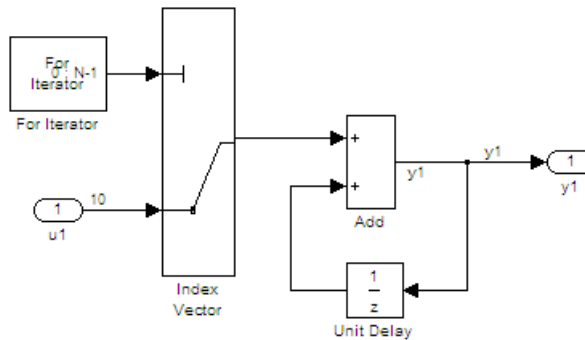
- “Modeling Pattern for For Loop: For-Iterator Subsystem block” on page 5-39
- “Modeling Pattern for For Loop: Stateflow Chart” on page 5-42
- “Modeling Pattern for For Loop: Embedded MATLAB block” on page 5-45

Modeling Pattern for For Loop: For-Iterator Subsystem block

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



Model For_Loop_SL



For Iterator Subsystem

Procedure.

- 1 Drag a For Iterator Subsystem block from the **Simulink > Ports and Subsystems** library into your model.
- 2 Connect the data inputs and outputs to the For Iterator Subsystem block.
- 3 Open the Inport block.
- 4 In the Block Parameters dialog box, select the **Signal Attributes** pane and set the **Port dimensions** parameter to 10.
- 5 Double-click the For Iterator Subsystem block to open the subsystem.

- 6 Drag an Index Vector block from the Signal-Routing library into the subsystem.
- 7 Open the For Iterator block. In the Block Parameters dialog box set the **Index-mode** parameter to Zero-based and the **Iteration limit** parameter to 10.
- 8 Connect the controlling input to the topmost input port of the Index Vector block, and the other input to the second port.
- 9 Drag an Add block from the **Math Operations** library into the subsystem.
- 10 Drag a Unit Delay block from **Commonly Used Blocks** library into the subsystem.
- 11 Double-click the Unit Delay block and set the **Initial Conditions** parameter to 0. This parameter initializes the state to zero.
- 12 Connect the blocks as shown in the model diagram.
- 13 Save the subsystem and the model.
- 14 Enter Ctrl+B to build the model and generate code.

Results. Real-Time Workshop software generates the following For_Loop_SL_step function in the file For_Loop_SL.c:

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

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

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

    /* Outputs for iterator SubSystem: '<Root>/For Iterator Subsystem' incorporates:
     * ForIterator: '<S1>/For Iterator'
    */
}
```

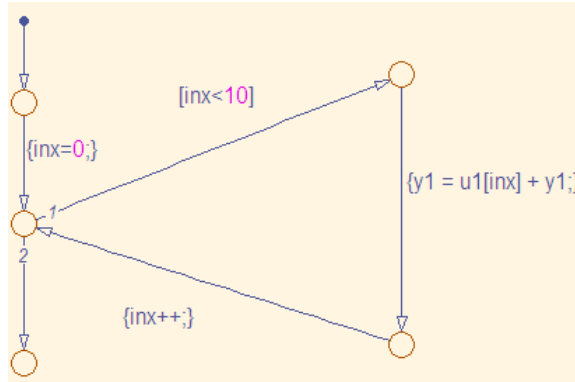
```
*/
for (s1_iter = 0; s1_iter < 10; s1_iter++) {
    /* Sum: '<S1>/Add' incorporates:
    * Inport: '<Root>/u1'
    * MultiPortSwitch: '<S1>/Index Vector'
    * UnitDelay: '<S1>/Unit Delay'
    */
    rtb_y1 = U.u1[s1_iter] + DWork.UnitDelay_DSTATE;

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

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

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

Modeling Pattern for For Loop: Stateflow Chart



Procedure.

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

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

The Stateflow diagram is shown.

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

Results. Real-Time Workshop software generates the following For_Loop_SF_step function in the file For_Loop_SF.c:

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

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

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

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

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

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

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

    /* Output: '<Root>/y1' */
}
```

```
Y.y1 = B.y1;  
}
```


Modeling Pattern for For Loop: Embedded MATLAB block

Procedure.

- 1 Follow the directions for “Setting Up an Example Model With an Embedded MATLAB Block” on page 5-7. This example model contains one Inport block and one Outport block.
- 2 Name your model For_Loop_EML.
- 3 In the Embedded MATLAB editor enter the function, as follows:

```
function y1 = fcn(u1)

y1 = 0;

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

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

Results. Real-Time Workshop software generates the following For_Loop_EML_step function in the file For_Loop_EML.c:

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

/* Model step function */
void For_Loop_EML_step(void)
{
    int32_T eml_inx;

    /* Embedded MATLAB: '<Root>/Embedded MATLAB Function' incorporates:
     * Inport: '<Root>/u1'
     */
    /* Embedded MATLAB Function 'Embedded MATLAB Function': '<S1>:1' */
    /* '<S1>:1:3' */
    y1 = 0.0;
    for (eml_inx = 0; eml_inx < 10; eml_inx++) {
```

```
        /* '<S1>:1:5' */
        /* '<S1>:1:6' */
        y1 = u1[eml_inx] + y1;
    }
}
```

While loop

C Construct

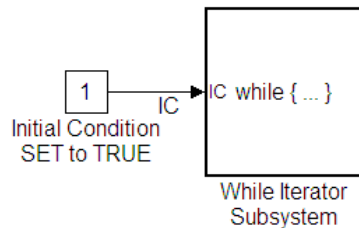
```
while(flag && (num_iter <= 100)
{
    flag = func ();
    num_iter ++;
}
```

Modeling Patterns

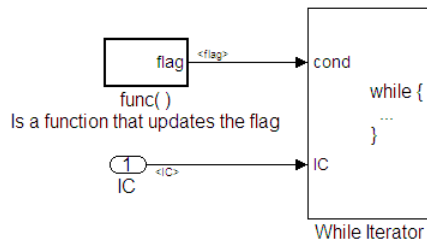
- “Modeling Pattern for While Loop: While Iterator Subsystem block” on page 5-47
- “Modeling Pattern for While Loop: Stateflow Chart” on page 5-51
- “Modeling Pattern for While Loop: Embedded MATLAB block” on page 5-54

Modeling Pattern for While Loop: While Iterator Subsystem block

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



Model While_Loop_SL

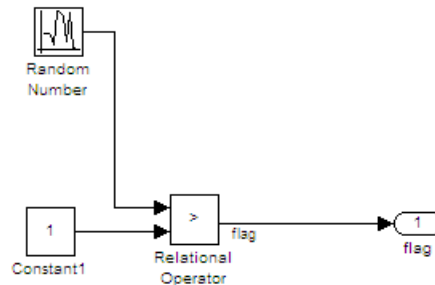


While_Loop_SL/While Iterator Subsystem

Procedure.

- 1** Drag a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library into the model.
- 2** Drag a Constant block from the **Simulink > Commonly Used Blocks** library into the model. In this case, set the **Initial Condition** to 1 and the **Data Type** to **Boolean**. You do not always have to set the initial condition to **FALSE**. The initial condition can be dependent on the input to the block.
- 3** Connect the Constant block to the While Iterator Subsystem block.
- 4** Double-click the While Iterator Subsystem block to open the subsystem.

- 5 Place a Subsystem block next to the While Iterator block.
- 6 Right-click the subsystem and select **Subsystem Parameters**. The Block Parameters dialog box opens.
- 7 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function.
- 8 From the **Real-Time Workshop system code** list, select the option, Function.
- 9 From the **Real-Time Workshop function name options** list, select the option, User specified. The **Real-Time Workshop function name** parameter is displayed.
- 10 Specify the name as func.
- 11 Click **Apply**.
- 12 Double-click the func() subsystem block. In this example, function func() has an output flag set to 0 or 1 depending on the result of the algorithm in func(). Create the func() algorithm as shown in the following diagram:



func()

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

Results. Real-Time Workshop software generates the following While_Loop_SL_step function in the file While_Loop_SL.c:

```

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

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

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

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

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

/* Model step function */
void While_Loop_SL_step(void)
{
    int32_T s1_iter;
    boolean_T loopCond;

    /* Outputs for iterator SubSystem:
     * '<Root>/While Iterator Subsystem' incorporates:

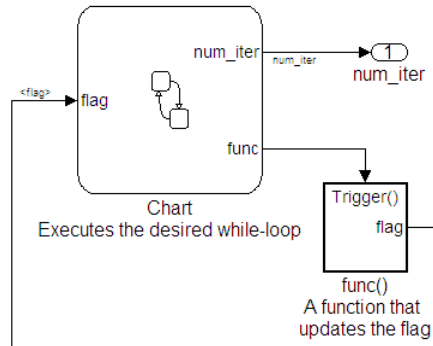
```

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

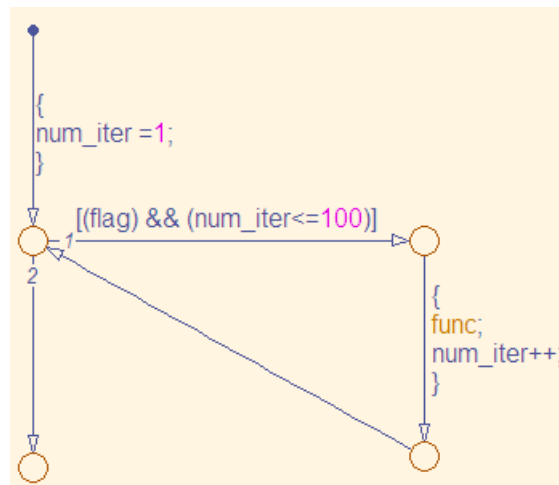
        /* end of Outputs for SubSystem:
        * '<S1>/func( ) Is a function that updates the flag' */
        loopCond = flag;
        s1_iter++;
    }

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

Modeling Pattern for While Loop: Stateflow Chart



Model While_Loop_SF



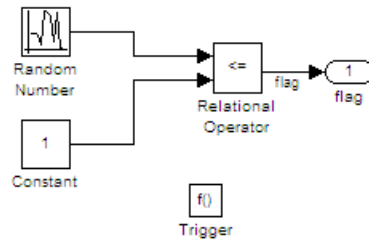
While_Loop_SF/Chart Executes the desired while-loop

Procedure.

- 1 Add a Stateflow Chart to your model from the **Stateflow > Stateflow Chart** library.
- 2 Double-click the chart and select **Tools > Explore** or enter **Ctrl+R** to open the Model Explorer.

- 3 Add the inputs and outputs to the chart and specify their data type.
- 4 Connect the data input and output to the Stateflow chart as shown in the model diagram.
- 5 In the Model Explorer, select the output variable, then, in the right pane, select the **Value Attributes** tab and set the **Initial Value** to 0.
- 6 Select **Patterns > Add Loop > While**. The Stateflow Pattern dialog opens.
- 7 Fill in the fields for the Stateflow Pattern dialog box as follows:

Description	While Loop (optional)
While condition	(flag) && (num_iter<=100)
Do action	func; num_iter++;
- 8 Place a Subsystem block in your model.
- 9 Right-click the subsystem and select Subsystem Parameters. The Block Parameters dialog box opens.
- 10 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function.
- 11 From the **Real-Time Workshop system code** list, select the option, Function.
- 12 From the **Real-Time Workshop function name options** list, select the option, User specified. The **Real-Time Workshop function name** parameter is displayed.
- 13 Specify the name as func.
- 14 Click **Apply** to apply all changes.
- 15 Double-click the func() subsystem block. In this example, function func() has an output flag set to 0 or 1 depending on the result of the algorithm in func(). The Trigger block parameter **Trigger type** is function-call. Create the func() algorithm, as shown in the following diagram:



While_Loop_SF/func() A function that updates the flag

16 Save and close the subsystem.

17 Connect blocks to the Stateflow chart as shown in the model diagram for While_Loop_SF.

18 Save your model.

Results. Real-Time Workshop software generates the following While_Loop_SF_step function in the file While_Loop_SF.c:

```

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

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

/* Model step function */
void While_Loop_SF_step(void)
{
    /* Stateflow: '<Root>/Chart Executes the desired
    * while-loop' incorporates:
    * SubSystem: '<Root>/func() A function that
    * updates the flag'
    */
    /* Gateway: Chart
    Executes the desired while-loop */
    /* During: Chart
    Executes the desired while-loop */
    /* Transition: '<S1>:2' */

```

```

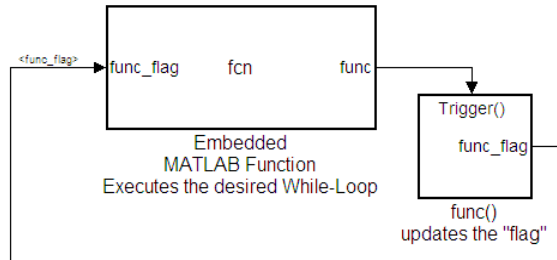
num_iter = 1;
while (flag && (num_iter <= 100)) {
    /* Transition: '<S1>:3' */
    /* Transition: '<S1>:4' */
    /* Event: '<S1>:12' */
    func();
    num_iter = num_iter + 1;

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

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

```

Modeling Pattern for While Loop: Embedded MATLAB block



Model While_Loop_EML

Procedure.

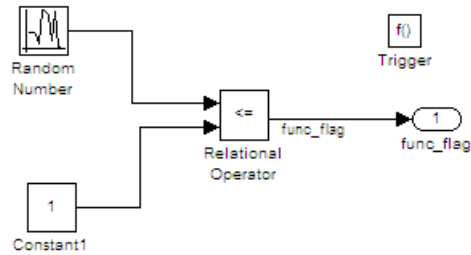
- 1 In the Simulink Library Browser, click **Simulink > User Defined Functions**, and drag an Embedded MATLAB Function block into your model.
- 2 Double-click the Embedded MATLAB Function block. The Embedded MATLAB editor opens.
- 3 In the Embedded MATLAB editor enter the function, as follows:

```
function fcn(func_flag)
```

```
flag = true;
num_iter = 1;

while(flag && (num_iter<=100))
    func;
    flag = func_flag;
    num_iter = num_iter + 1;
end
```

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



12 Save and close the subsystem.

13 Connect the Embedded MATLAB Function block to the `func()` subsystem.

14 Save your model.

Results. Real-Time Workshop software generates the following `While_Loop_EML_step` function in the file `While_Loop_EML.c`. In some cases an equivalent for loop might be generated instead of a while loop.

```

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

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

/* Model step function */
void While_Loop_EML_step(void)
{
    boolean_T eml_func_flag;
    boolean_T eml_flag;
    int32_T eml_num_iter;

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

    /* Embedded MATLAB Function 'Embedded MATLAB Function
    * Executes the desired While-Loop': '<S1>:1' */

```

```

/* '<S1>:1:3' */
eml_flag = true;

/* '<S1>:1:4' */
for (eml_num_iter = 1; eml_flag && (eml_num_iter <= 100);
    eml_num_iter++) {
/* '<S1>:1:6' */
/* '<S1>:1:7' */
func();

/* '<S1>:1:8' */
eml_flag = eml_func_flag;

/* '<S1>:1:9' */
}
}

```

Do While loop

C Construct

```

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

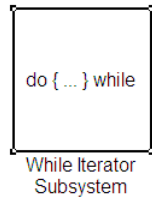
```

Modeling Patterns

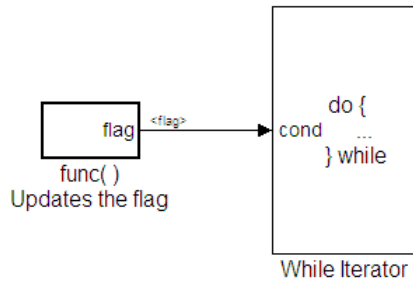
- “Modeling Pattern for Do While Loop: While Iterator Subsystem block” on page 5-58
- “Modeling Pattern for Do While Loop: Stateflow Chart” on page 5-61

Modeling Pattern for Do While Loop: While Iterator Subsystem block

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



Do_While_Loop_SL

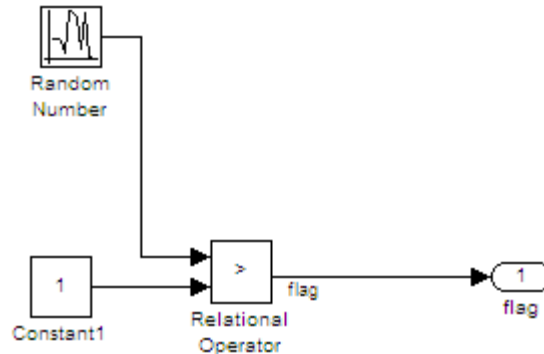


Do_While_Loop_SL/While Iterator Subsystem

Procedure.

- 1 Drag a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library into the model.
- 2 Double-click the While Iterator Subsystem block to open the subsystem.
- 3 Place a Subsystem block next to the While Iterator block.
- 4 Right-click the subsystem and select **Subsystem Parameters**. The Block Parameters dialog box opens.

- 5 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function.
- 6 From the **Real-Time Workshop system code** list, select the option, Function.
- 7 From the **Real-Time Workshop function name options** list, select the option, User specified. The **Real-Time Workshop function name** parameter is displayed.
- 8 Specify the name as func.
- 9 Click **Apply**.
- 10 Double-click the func() subsystem block. In this example, function func() has an output flag set to 0 or 1 depending on the result of the algorithm in func(). Create the func() algorithm as shown in the following diagram:



Do_While_Loop_SL/While Iterator Subsystem/func()

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

15 Enter Ctrl+B to generate code.

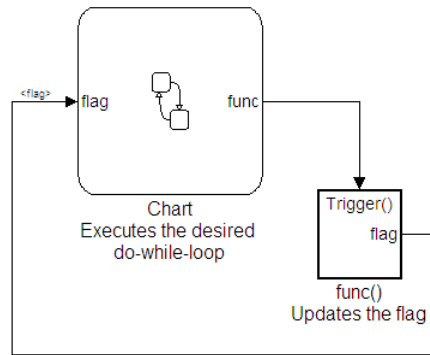
Results.

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

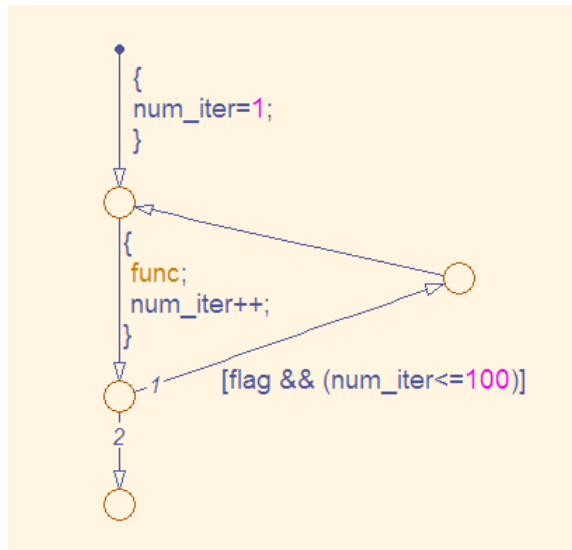
void Do_While_Loop_SL_step(void)
{
    int32_T s1_iter;

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


Modeling Pattern for Do While Loop: Stateflow Chart



Do_While_Loop_SF



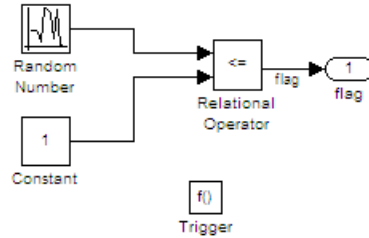
Do_While_Loop_SF/Chart Executes the desired do-while-loop

- 1 Add a Stateflow Chart to your model from the **Stateflow > Stateflow Chart** library.

- 2 Double-click the chart and select **Tools > Explore** or enter Ctrl+B to open the Model Explorer.
- 3 Add the inputs and outputs to the chart and specify their data type.
- 4 Connect the data input and output to the Stateflow chart.
- 5 In the Model Explorer, select the output variable, then, in the right pane, select the **Value Attributes** tab and set the **Initial Value** to 0.
- 6 Select **Patterns > Add Loop > While**. The Stateflow Pattern dialog opens.
- 7 Fill in the fields for the Stateflow Pattern dialog box as follows:

Description	While Loop (optional)
While condition	(flag) && (num_iter<=100)
Do action	func; num_iter++;
- 8 Place a Subsystem block in your model.
- 9 Right-click the subsystem and select Subsystem Parameters. The Block Parameters dialog box opens.
- 10 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function.
- 11 From the **Real-Time Workshop system code** list, select the option, Function.
- 12 From the **Real-Time Workshop function name options** list, select the option, User specified. The **Real-Time Workshop function name** parameter is displayed.
- 13 Specify the name as func.
- 14 Click **Apply** to apply all changes.
- 15 Double-click the func() subsystem block. In this example, function func() has an output flag set to 0 or 1 depending on the result of the algorithm

in `func()`. The Trigger block parameter **Trigger type** is `function-call`. Create the `func()` algorithm, as shown in the following diagram:



Do_While_Loop_SF/func() Updates the flag

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

Results.

```

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

Functions

In this section...

“Function Call” on page 5-64

“Function Prototyping” on page 5-66

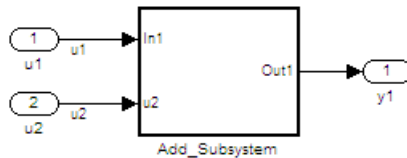
“External C Functions” on page 5-69

Function Call

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

C Construct

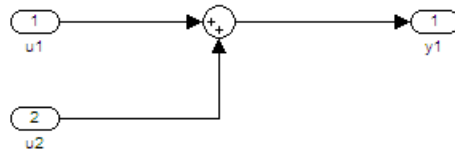
```
void add_function(void)
{
    y1 = u1 + u2;
}
```



Function_Call

Procedure

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



Function_Call/Add_Subsystem

3 Right-click the subsystem to open the Subsystem Parameters dialog box.

4 Select the **Treat as atomic unit** parameter.

For the **Real-Time Workshop system code** parameter, from the drop-down list, select **Function**.

5 For the **Real-Time Workshop function name options** parameter, from the drop-down list, select **User specified**.

6 In the **Real-Time Workshop function name** field, enter the subsystem name, `add_function`.

7 Click **Apply** and **OK**.

8 Press **Ctrl+B** to build and generate code.

Results

In `Function_call.c`, the function is called from `Function_Call_step`:

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

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

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

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

```
void add_function(void)
{
```

```
Function_Call_Y.y1 = u1 + u2;  
}
```

Function Prototyping

C Construct

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

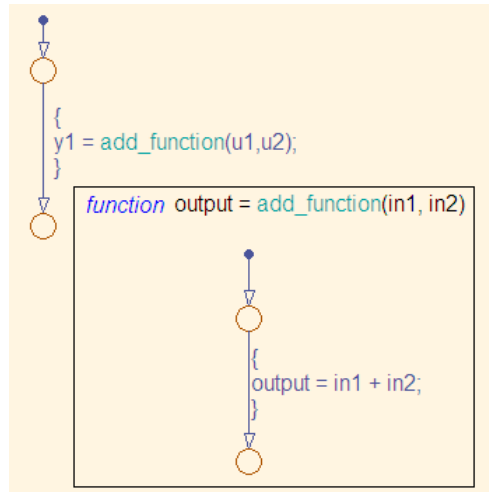
Modeling Patterns

- “Function Call Using Graphical Functions” on page 5-66
- “Control Function Prototype of the *model_step* Function” on page 5-68

Function Call Using Graphical Functions

Procedure.

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



Func_SF/Chart

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

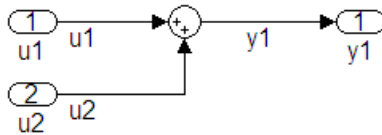
- 6 Open the Model Explorer. From the Model Hierarchy tree, select **Func_SF > Chart > f()add_function**. On the right pane, specify the **Function Inline Option** as **Function**.
- 7 From the Model Hierarchy tree, click **Chart** and on the right pane select the **Export Chart Level Graphical Functions(Make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

Results. `Func_SF.c` contains the generated code:

```
extern real_T add_function(real_T sf_in1, real_T sf_in2)
{
    return sf_in1 + sf_in2;
}
.
.
.
void Func_SF_step(void)
{
```

```
Func_SF_B.y1 = add_function(u1, u2);  
  
Func_SF_Y.y1 = Func_SF_B.y1;  
}
```

Control Function Prototype of the *model_step* Function



control_step_function

Procedure.

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

7 To validate your changes, click **Validate**.

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

Results. `control_step_function.c` contains the generated code:

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

External C Functions

C Construct

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

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

Modeling Patterns

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

- “Using the Legacy Code Tool to Create S-functions” on page 5-70
- “Using a Stateflow Chart to Make Calls to C Functions” on page 5-72

- “Using an Embedded MATLAB Block to Make Calls to C Functions” on page 5-74

Using the Legacy Code Tool to Create S-functions

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

Procedure.

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

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

- 2 Create a C source file named `add.c` that contains the function body:

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

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

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

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

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

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

```

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

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

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

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

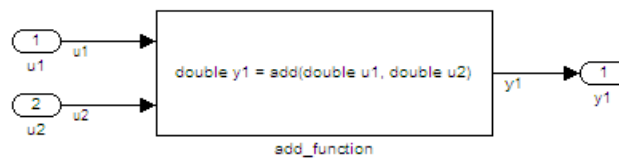
%% Create a Masked S-function Block
legacy_code('slblock_generate', def);

```

The output of this script produces:

- A new model containing the S-function block
- A TLC file named `add_function.tlc`.
- A C source file named `add_function.c`.
- A mexw32 dll file named `add_function.mexw32`

- 4 Add inport blocks and an output block and make the connections, as shown in the model.



Func_Call_LCT

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

Results. The following code is generated in `Function_Call_LCT.c`:

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

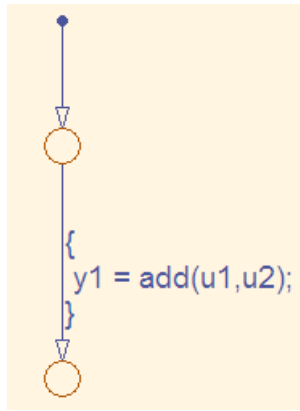
The user-specified header file, `add.h`, is included in `Function_Call_LCT.h`:

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

Using a Stateflow Chart to Make Calls to C Functions

Procedure.

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



ExFunction_Call_SF/Chart

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

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

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

Results. `ExFunction_Call_SF.c` contains the following code in the step function:

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

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

```
}
```

ExFunction_Call_SF.h contains the include statement for add.h:

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

Using an Embedded MATLAB Block to Make Calls to C Functions

Procedure.

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

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

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

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

end
```

- 6 In the Embedded MATLAB editor, select **Tools > Open Simulation Target**. The Configuration Parameters dialog box opens with the **Simulation Target** pane displayed.
- 7 On the left pane of the Configuration Parameters dialog box under **Simulation Target**, select **Custom Code**. In the **Include custom C**

code in generated section, on the left pane, select **Header file** and in the **Header file** field, enter the statement, :

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

- 8** In the **Include list of additional** section, select **Source files** and in the **Source files** field, enter `add.c`.
- 9** Add two Inport blocks and one Outport block to the model and connect to the Embedded MATLAB block.
- 10** Configure the signals: `u1`, `u2`, and `y1`, as described in “Configuring a Signal” on page 5-3.
- 11** Save the model as `ExFunction_Call_EML.mdl`.
- 12** Press **Ctrl+B** to build the model and generate code.

Results. `ExFunction_Call_EML.c` contains the following code:

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

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

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

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


Defining Data Representation and Storage for Code Generation

- Chapter 6, “Using mpt Data Objects”
- Chapter 7, “Creating and Using Custom Storage Classes”
- Chapter 8, “Inserting Comments and Pragmas in Generated Code”
- Chapter 9, “Optimizing Buses for Code Generation”
- Chapter 10, “Renaming and Replacing Data Types”
- Chapter 11, “Managing Data Definitions and Declarations With the Data Dictionary”
- Chapter 12, “Managing Placement of Data Definitions and Declarations”
- Chapter 13, “Specifying the Persistence Level for Signals and Parameters”

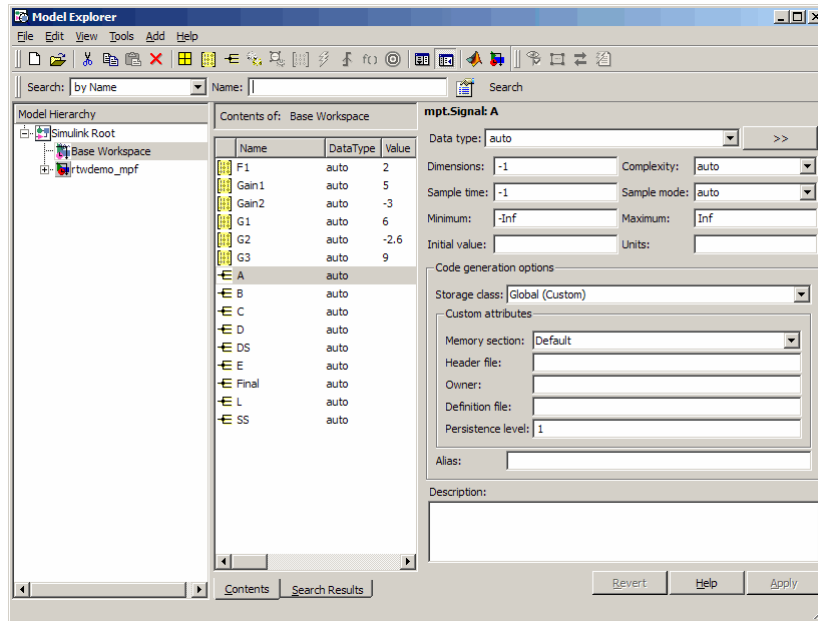
Using mpt Data Objects

The following table describes the properties and property values for all `mpt.Parameter` and `mpt.Signal` data objects that appear in the Model Explorer.

Note You can create `mpt.Signal` and `mpt.Parameter` objects in the base MATLAB or model workspace. However, if you create the object in a model workspace, the object's storage class must be set to `auto`.

The figure below shows an example of the Model Explorer. When you select an `mpt.Parameter` or `mpt.Signal` data object in the middle pane, its properties and property values display in the rightmost pane.

In the Properties column, the table lists the properties in the order in which they appear on the Model Explorer. Another table describes the effects that example changes to property values have on the generated code.



Parameter and Signal Property Values

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	User object type	*auto	<p>Prenamed and predefined property sets that are registered in the <code>sl_customization.m</code> file. (See “Registering mpt User Object Types” on page 11-48.) This field is unavailable if no user object type is registered.</p> <p>Select auto if this field is available but you do not want to apply the properties of a user object type to a selected data object. The fields on the Model Explorer are populated with default values.</p>

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
		Any user object type name listed	Select a user object type name to apply the properties and values that you associated with this name in the <code>sl_customization.m</code> file. The fields on the Model Explorer are automatically populated with those values.
Parameter	Value	*0	The data type and numeric value of the data object. For example, <code>int8(5)</code> . The numeric value is used as an initial parameter value in the generated code.
Both	Data type		Used to specify the data type for an <code>mpt.Signal</code> data object, but not for an <code>mpt.Parameter</code> data object. The data type for an <code>mpt.Parameter</code> data object is specified in the Value field above. See “Working with Data Types” in the Simulink documentation.
Both	Units	*null	Units of measurement of the signal or parameter. (Enter text in this field.)
Both	Dimensions	* -1	The dimension of the signal or parameter. For a parameter, the dimension is derived from its value.
Both	Complexity	*auto real complex	Complexity specifies whether the signal or parameter is a real or complex number. Select <code>auto</code> for the code generator to decide. For a parameter, the complexity is derived from its value.
Signal	Sample time	* -1	Model or block execution rate.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Signal	Sample mode	*auto	Determines how the signal propagates through the model. Select auto for the code generator to decide.
		Sample based	The signal propagates through the model one sample at a time.
		Frame based	The signal propagates through the model in batches of samples.
Both	Minimum	*0.0	The minimum value to which the parameter or signal is expected to be bound.
		Any number within the minimum range of the parameter or signal. (Based on the data type and resolution of the parameter or signal.)	
Both	Maximum	*0.0	Maximum value to which the parameter or signal is expected to be bound. (Enter information using a dialog box.)
	Code generation options		
	Storage class		Note that an auto selection for a storage class tells the Real-Time Workshop build process to decide how to declare and store the selected parameter or signal.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Default (Custom)		Real-Time Workshop Embedded Coder software decides how to declare the data object.
Both	Global (Custom)	Global (Custom) is the default storage class for mpt data objects.	Ensures that the code generator places no qualifier in the data object's declaration.
Both	Memory section	*Default	Memory section allows you to specify storage directives for the data object. Default ensures that the code generator places no type qualifier and no <code>pragma</code> statement with the data object's declaration.
Parameter		MemConst	Places the <code>const</code> type qualifier in the declaration.
Both		MemVolatile	Places the <code>volatile</code> type qualifier in the declaration.
Parameter		MemConstVolatile	Places the <code>const volatile</code> type qualifier in the declaration.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Header file		<p>Name of the file used to import or export the data object. This file contains the declaration (<code>extern</code>) to the data object.</p> <p>Also, you can specify this header filename between the double-quotation or angle-bracket delimiter. You can specify the delimiter with or without the <code>.h</code> extension. For example, "<code>object.h</code>" or "<code>object</code>" has the same effect. For the selected data object, this overrides the general delimiter selection in the #include file delimiter field on the Configuration Parameters dialog box.</p>
Both	Owner	*Blank	<p>The name of the module that owns this signal or parameter. This is used to help determine the ownership of a definition. For details, see "Ownership Settings" on page 12-10 and "Effects of Ownership Settings" on page 12-22.</p>
Both	Definition file	*Blank	<p>Name of the file that defines the data object.</p>
		Any valid text string	

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Persistence level		The number you specify is relative to Signal display level or Parameter tune level on the Data Placement pane of the Configuration Parameters dialog box. For a signal, allows you to specify whether or not the code generator declares the data object as global data. For a parameter, allows you to specify whether or not the code generator declares the data object as tunable global data. See Signal display level and Parameter tune level in “Real-Time Workshop Pane: Code Placement”.
Both	Bitfield (Custom)		Embeds Boolean data in a named bit field.
	Struct name		Name of the struct into which the object’s data will be packed.
Parameter	Const (Custom)		Places the const type qualifier in the declaration.
Parameter	Header file		See above.
Parameter	Owner		See above.
Parameter	Definition file		See above.
Parameter	Persistence level		See above.
Both	Volatile (Custom)		Places the volatile type qualifier in the declaration.
Both	Header file		See above.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Owner		See above.
Both	Definition file		See above.
Both	Persistence level		See above.
Parameter	ConstVolatile (Custom)		Places the <code>const volatile</code> type qualifier in declaration.
Parameter	Header file		See above.
Parameter	Owner		See above.
Parameter	Definition file		See above.
Parameter	Persistence level		See above.
Parameter	Define (Custom)		Represents parameters with a <code>#define</code> macro.
Parameter	Header file		See above.
Both	ExportToFile (Custom)		Generates global variable definition, and generates a user-specified header (.h) file that contains the declaration (<code>extern</code>) to that variable.
Both	Memory section		See above.
Both	Header file		See above.
Both	Definition file		See above.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	ImportFromFile (Custom)		Includes predefined header files containing global variable declarations, and places the #include in a corresponding file. Assumes external code defines (allocates memory) for the global variable.
Both	Data access	*Direct	Allows you to specify whether the identifier that corresponds to the selected data object stores data of a data type (Direct) or stores the address of the data (a pointer).
Both		Pointer	If you select Pointer, the code generator places * before the identifier in the generated code.
	Header file		See above.
Both	Struct (Custom)		Embeds data in a named struct to encapsulate sets of data.
Both	Struct name		See above.
Signal	GetSet (Custom)		Reads (gets) and writes (sets) data using functions.
Signal	Header file		See above.
Signal	Get function		Specify the Get function.
Signal	Set function		Specify the Set function.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Alias	*null	As explained in detail in “Applying Naming Rules to Identifiers Globally” on page 11-30, for a Simulink or mpt data object (identifier), specifying a name in the Alias field overrides the global naming rule selection you make on the Configuration Parameters dialog box.
		Any valid ANSI ¹ C/C++ variable name	
Both	Description	*null	Text description of the parameter or signal. Appears as a comment beside the signal or parameter’s identifier in the generated code.
		Any text string	

1. ANSI[®] is a registered trademark of the American National Standards Institute, Inc.

Some Examples of the Effect of Property Value Changes on Generated Code

What I noticed when inspecting the .c/.cpp file	Change I made to property value settings	What I noticed after regenerating and reinspecting the file
<p>Example 1: Parameter data objects can be declared or defined as constants. I know that the data object GAIN is a parameter. I want this to be declared or defined in the .c file as a variable. But I notice that GAIN is declared as a constant by the statement <code>const real_T GAIN = 5.0;</code>. Also, this statement is in the constant section of the file.</p>	<p>In the Model Explorer, I clicked the data object GAIN. I noticed that the property value for its Memory section property is set at MemConst. I changed this to Default.</p>	<p>I notice two differences. One is that now GAIN is declared as a variable with the statement <code>real_T GAIN = 5.0;</code>. The second difference is that the declaration now is located in the MemConst memory section in the .c or .cpp file.</p>
<p>Example 2: I notice again the declaration of GAIN in the .c file mentioned in Example 1. It appears as <code>real_T GAIN = 5.0;</code>. But I have changed my mind. I want data object GAIN to be <code>#define</code>.</p>	<p>I changed the Storage class selection to Define (Custom).</p>	<p>GAIN is no longer declared in the .c file as a MemConst parameter. Rather, it now is defined as a <code>#define</code> macro by the code <code>#define GAIN 5.0</code>, and this is located near the top of the .c file with the other preprocessor directives.</p>

Some Examples of the Effect of Property Value Changes on Generated Code (Continued)

What I noticed when inspecting the .c/.cpp file	Change I made to property value settings	What I noticed after regenerating and reinspecting the file
<p>Example 3: I changed my mind again after doing Example 2. I do want GAIN defined using the <code>#define</code> preprocessor directive. But I do not want to include the <code>#define</code> in this file. I know it exists in another file and I want to reference that file.</p>	<p>On the Model Explorer, I notice that the property value for the Header file property is blank. I changed this to <code>filename.h</code>. (I chose the ANSI C/C++ double quote mechanism for the <code>#include</code>, but could have chosen the angle bracket mechanism.) Also, it is necessary that I make the user-defined <code>filename.h</code> available to the compiler, placing it either in the system path or local directory.</p>	<p>The <code>#define GAIN 5.0</code> is no longer in this .c file. Instead, the <code>#include filename.h</code> code appears as a preprocessor directive at the top of the file.</p>
<p>Example 4: I have one more change I want to make. Let us say that we have declared the data object <code>data_in</code>, and that its declaration statement in the .c file reads <code>real_T data_in = 0.0;</code>. I want to replace this in all locations in the .c file with an alias.</p>	<p>In the Model Explorer, I selected the data object <code>data_in</code>. I noticed that the Alias field is blank. I changed this to <code>data_in_alias</code>, which I know is a valid ANSI C/C++ variable name.</p>	<p>The identifier <code>data_in_alias</code> now appears in the .c file everywhere <code>data_in</code> appeared.</p>

Creating and Using Custom Storage Classes

- “Introduction to Custom Storage Classes” on page 7-2
- “Resources for Defining Custom Storage Classes” on page 7-5
- “Simulink Package Custom Storage Classes” on page 7-6
- “Creating Packages that Support CSC Definitions” on page 7-8
- “Designing Custom Storage Classes and Memory Sections” on page 7-12
- “Applying CSCs to Parameters and Signals” on page 7-37
- “Generating Code with Custom Storage Classes” on page 7-58
- “Defining Advanced Custom Storage Class Types” on page 7-62
- “GetSet Custom Storage Class for Data Store Memory” on page 7-66
- “Custom Storage Class Implementation” on page 7-70
- “Custom Storage Class Limitations” on page 7-72
- “Custom Storage Classes Prior to R2009a” on page 7-73
- “Custom Storage Classes Prior to Release 14” on page 7-74

Introduction to Custom Storage Classes

In this section...

“Custom Storage Class Memory Sections” on page 7-3

“Registering Custom Storage Classes” on page 7-3

“Custom Storage Class Demos” on page 7-4

During the Real-Time Workshop build process, the *storage class* specification of a signal, tunable parameter, block state, or data object specifies how that entity is declared, stored, and represented in generated code. Note that in the context of the Real-Time Workshop build process, the term “storage class” is not synonymous with the term “storage class specifier”, as used in the C language.

The Real-Time Workshop software defines four built-in storage classes for use with all targets: `Auto`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. These storage classes provide limited control over the form of the code generated for references to the data. For example, data of storage class `Auto` is typically declared and accessed as an element of a structure, while data of storage class `ExportedGlobal` is declared and accessed as unstructured global variables. For information about built-in storage classes, see “Signal Considerations” and “Simulink Data Object Considerations” in the Real-Time Workshop documentation.

The built-in storage classes are suitable for many applications, but embedded system designers often require greater control over the representation of data. Real-Time Workshop Embedded Coder *custom storage classes* (CSCs) extend the built-in storage classes provided by the Real-Time Workshop software. CSCs can provide application-specific control over the constructs required to represent data in an embedded algorithm. For example, you can use CSCs to:

- Define structures for storage of parameter or signal data.
- Conserve memory by storing Boolean data in bit fields.
- Integrate generated code with legacy software whose interfaces cannot be modified.

- Generate data structures and definitions that comply with your organization’s software engineering guidelines for safety-critical code.

Custom storage classes affect only code generated for ERT targets. When **Configuration Parameters > Real-Time Workshop > Target Selection > System target file** specifies a GRT target, the names of custom storage classes sometimes appear in dialog boxes, but selecting a CSC is functionally the same as selecting Auto. See “Selecting and Configuring a Target ” for information about ERT and GRT targets.

Custom Storage Class Memory Sections

Every custom storage class has an associated *memory section* definition. A memory section is a named collection of properties related to placement of an object in memory; for example, in RAM, ROM, or flash memory. Memory section properties let you specify storage directives for data objects. For example, you can specify const declarations, or compiler-specific #pragma statements for allocation of storage in ROM or flash memory sections.

See “Creating and Editing Memory Section Definitions” on page 7-31 for details about using the Custom Storage Class designer to define memory sections. While memory sections are often used with data in custom storage classes, they can also be used with various other constructs. See Chapter 8, “Inserting Comments and Pragmas in Generated Code” for more information about using memory sections with custom storage classes, and complete information about using memory sections with other constructs.

Registering Custom Storage Classes

CSCs are associated with Simulink data class packages (such as the Simulink package) and with classes within packages (such as the Simulink.Parameter and Simulink.Signal classes). The custom storage classes associated with a package are defined by a *CSC registration file*. For example, a CSC registration file exists for the Simulink package. This registration file provides predefined CSCs for use with the Simulink.Signal and Simulink.Parameter classes, and with subclasses derived from these classes. The predefined CSCs are sufficient for a wide variety of applications.

If you use only predefined CSCs, you do not need to be concerned with CSC registration files. You cannot add or change CSCs associated with built-in

packages and classes, but you can create your own packages and subclasses, then associate CSCs with those. See “Custom Storage Class Implementation” on page 7-70 for more information.

Custom Storage Class Demos

Three demos are available that show Custom Storage Class capabilities:

`rtwdemo_cscpredef` — Shows predefined custom storage classes and embedded signal objects

`rtwdemo_importstruct` — Shows custom storage classes used to access imported data efficiently

`rtwdemo_advsc` — Shows how custom storage classes can support data dictionary driven modeling

To launch a demo, click the demo’s name above, or type its name in the MATLAB Command Window.

Resources for Defining Custom Storage Classes

The resources for working with custom storage class definitions are:

- The Simulink Data Class Designer, which you can use to create a data object package and enable the ability to define your own CSC definitions for classes contained in the package. For information about the Data Class Designer, see “Subclassing Simulink Data Classes” and “Creating Packages that Support CSC Definitions” on page 7-8.
- A set of ready-to-use CSCs. These CSCs are designed to be useful in code generation for embedded systems development. CSC functionality is integrated into the `Simulink.Signal` and `Simulink.Parameter` classes; you do not need to use special object classes to generate code with CSCs. If you are unfamiliar with the `Simulink.Signal` and `Simulink.Parameter` classes and objects, read the “Simulink Data Object Considerations” section of the Real-Time Workshop documentation.
- The Custom Storage Class Designer (`cscdesigner`) tool, which is described in this chapter. This tool lets you define CSCs that are tailored to your code generation requirements. The Custom Storage Class Designer provides a graphical user interface that you can use to implement CSCs. You can use your CSCs in code generation immediately, without any Target Language Compiler (TLC) or other programming. See “Designing Custom Storage Classes and Memory Sections” on page 7-12 for details.

Simulink Package Custom Storage Classes

The Simulink package includes a set of built-in custom storage classes. These are categorized as custom storage classes, even though they are built-in, because they:

- Extend the storage classes provided by the Real-Time Workshop software
- Are functionally the same as if you had defined them yourself using the CSC Designer

The MathWorks™ provides the built-in CSCs because they provide specifications that many users need. Otherwise, all such users would have to define the needed CSCs manually, causing a needless duplication of effort.

You cannot change the CSCs built into the Simulink package, but you can subclass the package and add CSCs to the subclass, following the steps in “Resources for Defining Custom Storage Classes” on page 7-5.

Some CSCs in the Simulink package are valid for parameter objects but not signal objects and vice versa. For example, you can assign the storage class `Const` to a parameter but not to a signal, because signal data is not constant. The next table defines the CSCs built into the Simulink package and shows where each of the CSCs can be used.

CSC Name	Purpose	Signals?	Parameters?
BitField	Generate a struct declaration that embeds Boolean data in named bit fields.	Y	Y
CompilerFlag	Supports preprocessor conditionals defined via compiler flag. See “Generating Code Variants for Variant Models” on page 3-2.	N	Y
Const	Generate a constant declaration with the <code>const</code> type qualifier.	N	Y
ConstVolatile	Generate declaration of volatile constant with the <code>const volatile</code> type qualifier.	N	Y

CSC Name	Purpose	Signals?	Parameters?
Default	Default is a placeholder CSC that the code generator assigns to the <code>RTWInfo.CustomStorageClass</code> property of signal and parameter objects when they are created. You cannot edit the Default CSC definition.	Y	Y
Define	Generate <code>#define</code> directive.	N	Y
ExportToFile	Generate header (.h) file, with user-specified name, containing global variable declarations.	Y	Y
GetSet	Supports specialized function calls to read and write the memory associated with a Data Store Memory block. See “GetSet Custom Storage Class for Data Store Memory” on page 7-66.	Y	Y
ImportedDefine	Supports preprocessor conditionals defined via legacy header file. See “Generating Code Variants for Variant Models” on page 3-2.	N	Y
ImportFromFile	Generate directives to include predefined header files containing global variable declarations.	Y	Y
Struct	Generate a <code>struct</code> declaration encapsulating parameter or signal object data.	Y	Y
Volatile	Use <code>volatile</code> type qualifier in declaration.	Y	Y

Creating Packages that Support CSC Definitions

You can create a package and associate your own CSC definitions with classes contained in the package. You do this by creating a data object package containing classes derived from `Simulink.Parameter` or `Simulink.Signal`. The procedure below shows how to create and configure such a package. For additional information, see “Subclassing Simulink Data Classes”.

- 1 Open the Simulink Data Class Designer by choosing **Tools > Data Class Designer** in the model window, or typing the following at the MATLAB command prompt:

```
sldataclassdesigner
```

- 2 The Data Class Designer loads all packages that exist on the MATLAB path.
- 3 To create a new package, click **New** next to the **Package name** field. If desired, edit the **Package name**. Then, click **OK**.
- 4 In the **Parent directory** field, enter the path to the directory where you want to store the new package.

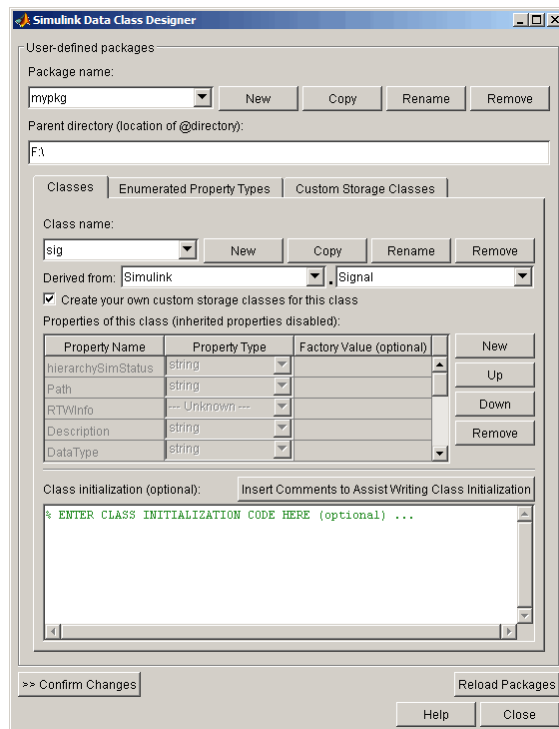
Note Do not create class package directories under *matlabroot*. Packages in these directories are treated as built-in and will not be visible in the Data Class Designer.

- 5 Click on the **Classes** tab.
- 6 Create a new class by clicking **New** next to the **Class name** field. If desired, edit the **Class name**. Then, click **OK**.
- 7 In the **Derived from** menu, select `Simulink.Signal` or `Simulink.Parameter`.
- 8 The **Create your own custom storage classes for this class** option is now enabled. This option is enabled when the selected class is derived from `Simulink.Signal` or `Simulink.Parameter`. You must select this option to create CSCs for the new class. If the **Create your own custom storage**

classes for this class option is not selected, the new class inherits the CSCs of the parent class.

Note To create a CSC registration file for a package, the **Create your own custom storage classes for this class** option must be selected for at least one of the classes in the package.

In the figure below, a new package called mypkg has been created. This package contains a new class, derived from Simulink.Signal, called sig. The **Create your own custom storage classes for this class** option is selected.



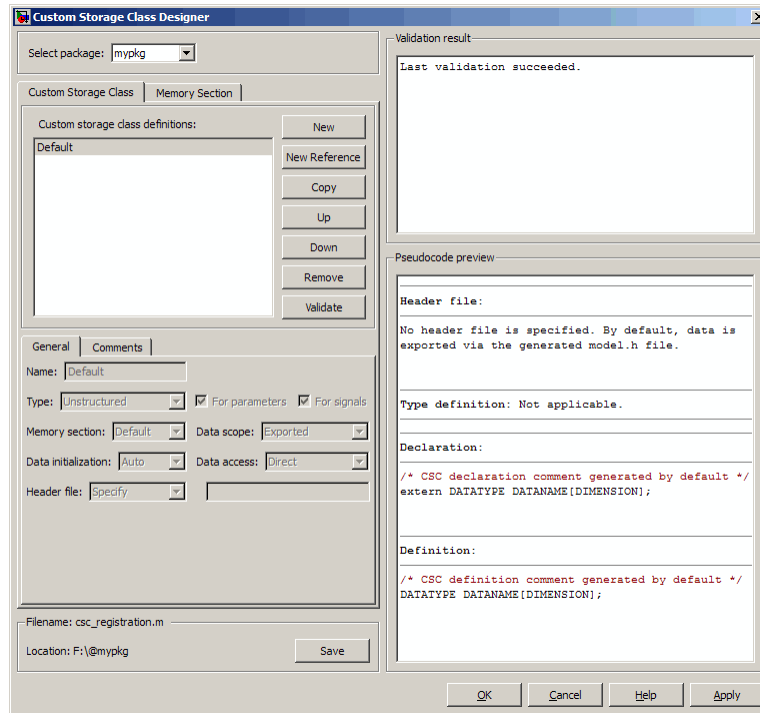
Complete instructions for using the Data Class Designer appear in “Subclassing Simulink Data Classes” in the Simulink documentation. See

also the instructions that appear when you click the **Custom Storage Classes** tab.

- 9 If desired, repeat steps 6–8 to add other derived classes to the package and associate CSCs with them.
- 10 Click **Confirm Changes**. In the **Confirm Changes** pane, select the package you created. Add the parent directory to the MATLAB path if necessary. Then, click **Write Selected**.

The package directories and files, including the CSC registration file, are written out to the parent directory.

- 11 Click **Close**.
- 12 You can now view and edit the CSCs belonging to your package in the Custom Storage Class Designer, which you open with the MATLAB command `cscdesigner`. Initially, the package contains only the Default CSC definition, as shown in the figure below.



- 13** Add and edit your CSC and memory section definitions, as described in “Designing Custom Storage Classes and Memory Sections” on page 7-12. After you have created CSC definitions for your package, you can instantiate objects of the classes belonging to your package, and assign CSCs to them.

You need to restart your MATLAB session before you can use the new CSCs with objects of your new classes.

Designing Custom Storage Classes and Memory Sections

In this section...

“Using the Custom Storage Class Designer” on page 7-12

“Editing Custom Storage Class Properties” on page 7-19

“Using Custom Storage Class References” on page 7-26

“Creating and Editing Memory Section Definitions” on page 7-31

“Using Memory Section References” on page 7-34

Using the Custom Storage Class Designer

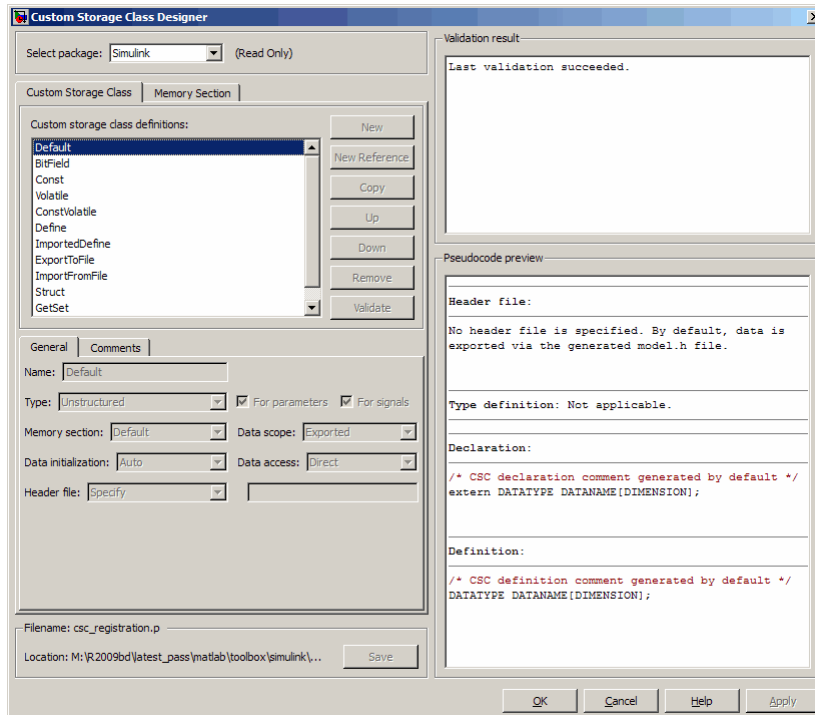
The Custom Storage Class Designer (`cscdesigner`) is a tool for creating and managing custom storage classes and memory sections. You can use the Custom Storage Class Designer to:

- Load existing custom storage classes and memory sections and view and edit their properties
- Create new custom storage classes and memory sections
- Create references to custom storage classes and memory sections defined in other packages
- Copy and modify existing custom storage class and memory section definitions
- Verify the correctness and consistency of custom storage class and memory section definitions
- Preview pseudocode generated from custom storage class and memory section definitions
- Save custom storage class and memory section definitions

To open the Custom Storage Class Designer, type the following command at the MATLAB prompt:

```
cscdesigner
```

When first opened, the Custom Storage Class Designer scans all data class packages on the MATLAB path to detect packages that have a CSC registration file. A message is displayed while scanning proceeds. When the scan is complete, the Custom Storage Class Designer window appears:



The Custom Storage Class Designer window is divided into several panels:

- **Select package:** Lets you select from a menu of data class packages that have CSC definitions associated with them. See “Selecting a Data Class Package” on page 7-14 for details.
- **Custom Storage Class / Memory Section** properties: Lets you select, view, edit, copy, verify, and perform other operations on CSC definitions or memory section definitions. The common controls in the **Custom Storage Class / Memory Section** properties panel are described in “Selecting and Editing CSCs, Memory Sections, and References” on page 7-15.

- When the **Custom Storage Class** tab is selected, you can select a CSC definition or reference from a list and edit its properties. See “Editing Custom Storage Class Properties” on page 7-19 for details.
- When the **Memory Section** tab is selected, you can select a memory section definition or reference from a list and edit its properties. See “Creating and Editing Memory Section Definitions” on page 7-31 for details.
- **Filename:** Displays the filename and location of the current CSC registration file, and lets you save your CSC definition to that file. See “Saving Your Definitions” on page 7-18 for details.
- **Pseudocode preview:** Displays a preview of code that is generated from objects of the given class. The preview is pseudocode, since the actual symbolic representation of data objects is not available until code generation time. See “Previewing Generated Code” on page 7-33 for details.
- **Validation result:** Displays any errors encountered when the currently selected CSC definition is validated. See “Validating CSC Definitions” on page 7-25 for details.

Selecting a Data Class Package

A CSC or memory section definition or reference is uniquely associated with a Simulink data class package. The link between the definition/reference and the package is formed when a CSC registration file (`csc_registration.m`) is located in the package directory.

You never need to search for or edit a CSC registration file directly: the Custom Storage Class Designer locates all available CSC registration files. The **Select package** menu contains names of all data class packages that have a CSC registration file on the MATLAB search path. At least one such package, the Simulink package, is always present.

When you select a package, the CSCs and memory section definitions belonging to the package are loaded into memory and their names are displayed in the scrolling list in the **Custom storage class** panel. The name and location of the CSC registration file for the package is displayed in the **Filename** panel.

If you select a user-defined package, by default you can use the Custom Storage Class Designer to edit its custom storage classes and memory sections. If you select a built-in package, you cannot edit its custom storage classes or memory sections. See “Custom Storage Class Implementation” on page 7-70 for more information.

Selecting and Editing CSCs, Memory Sections, and References

The **Custom Storage Class / Memory Section** panel lets you select, view, and (if the CSC is writable) edit CSC and memory section definitions and references. In the next figure and the subsequent examples, the selected package is `mypkg`. Instructions for creating a user-defined package like `mypkg` appear in “Creating Packages that Support CSC Definitions” on page 7-8.

The screenshot shows the Custom Storage Class Designer interface for the package `mypkg`. The interface is divided into several sections:

- Select package:** A dropdown menu showing `mypkg`.
- Custom Storage Class / Memory Section:** Two tabs are visible, with `Memory Section` selected.
- Custom storage class definitions:** A list box containing `Default`. To the right are buttons for `New`, `New Reference`, `Copy`, `Up`, `Down`, `Remove`, and `Validate`.
- General / Comments:** Two tabs are visible, with `General` selected. It contains the following fields:
 - Name:** `Default`
 - Type:** `Unstructured` (dropdown), with checkboxes for `For parameters` and `For signals` (both checked).
 - Memory section:** `Default` (dropdown)
 - Data scope:** `Exported` (dropdown)
 - Data initialization:** `Auto` (dropdown)
 - Data access:** `Direct` (dropdown)
 - Header file:** `Specify` (dropdown) and an empty text field.
- Filename:** `csc_registration.m`
- Location:** `F:\@mypkg` and a `Save` button.

The list at the top of the panel displays the definitions/references for the currently selected package. To select a definition/reference for viewing and editing, click on the desired list entry. The properties of the selected definition/reference appear in the area below the list. The number and type of properties vary for different types of CSC and memory section definitions. See:

- “Editing Custom Storage Class Properties” on page 7-19 for information about the properties of the predefined CSCs.
- “Creating and Editing Memory Section Definitions” on page 7-31 for information about the properties of the predefined memory section definitions.

The buttons to the right of the list perform these functions, which are common to both custom storage classes and memory definitions:

- **New:** Creates a new CSC or memory section with default values.
- **New Reference:** Creates a reference to a CSC or memory section definition in another package. The default initially has a default name and properties. See “Using Custom Storage Class References” on page 7-26 and “Using Memory Section References” on page 7-34.
- **Copy:** Creates a copy of the selected definition / reference. The copy initially has a default name using the convention:

`definition_name_n`

where `definition_name` is the name of the original definition, and `n` is an integer indicating successive copy numbers (for example: `BitField_1`, `BitField_2`, ...)

- **Up:** Moves the selected definition one position up in the list.
- **Down:** Moves the selected definition one position down in the list.
- **Remove:** Removes the selected definition from the list.
- **Validate:** Performs a consistency check on the currently selected definition. Errors are reported in the **Validation result** panel.

For example, if you click **New**, a new custom storage class is created with a default name:

Select package: mypkg

Custom Storage Class | Memory Section

Custom storage class definitions:

Default	New
NewCSC_1	New Reference
	Copy
	Up
	Down
	Remove
	Validate

General | Comments

Name: NewCSC_1

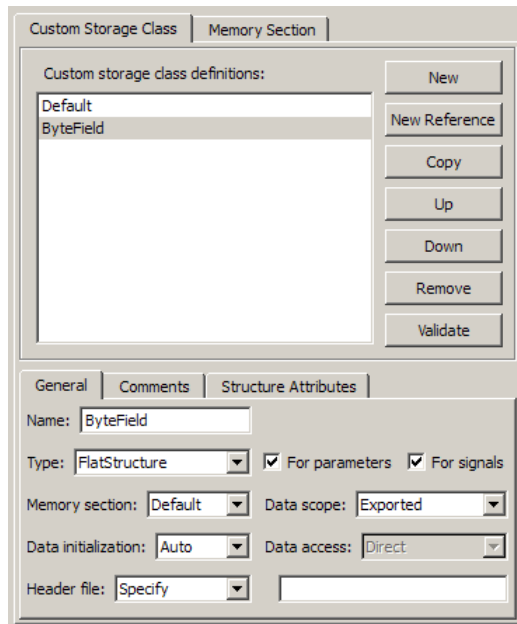
Type: Unstructured For parameters For signals

Memory section: Default Data scope: Auto

Data initialization: Auto Data access: Direct

Header file: Specify

You can now rename the new class by typing the desired name into the **Name** field, and set other fields as needed. The changes take effect when you click **Apply** or **OK**. For example, you could set values for the new custom storage class as follows:



Saving Your Definitions

After you have created or edited a CSC or memory section definition or reference, you must save the changes to the CSC registration file. To do this, click **Save** in the **Filename** panel. When you click **Save**, the current CSC and memory section definitions that are in memory are validated, and the definitions are written out.

If errors occur, they are reported in the **Validation result** panel. The definitions are saved whether or not errors exist. However, you should correct any validation errors and resave your definitions. Trying to use definitions that were saved with validation errors can cause additional errors. Such problems can occur even if you do not try to use the specific parts of the definition that contain the validation errors, making the problems difficult to diagnose.

Restarting MATLAB After Changing Definitions

If you add, change, or delete custom storage class or memory section definitions for any user-defined class, and objects of that class already exist,

you must restart MATLAB to put the changed definitions into effect and eliminate obsolete objects. A message warning you to restart MATLAB appears when you save the changed definitions. This warning message does not affect the success of the save operation itself.

Editing Custom Storage Class Properties

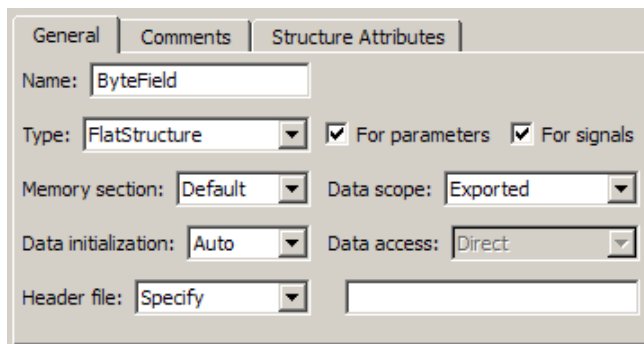
To view and edit the properties of a CSC, click the **Custom Storage Class** tab in the **Custom Storage Class / Memory Section** panel. Then, select a CSC name from the **Custom storage class definitions** list.

The CSC properties are divided into several categories, selected by tabs. Selecting a class, and setting property values for that class, can change the available tabs, properties, and values. As you change property values, the effect on the generated code is immediately displayed in the **Pseudocode preview** panel. In most cases, you can define your CSCs quickly and easily by selecting the **Pseudocode preview** panel and using the **Validate** button frequently.

The property categories and corresponding tabs are as follows:

General

Properties in the **General** category are common to all CSCs. In the next figure and the subsequent examples, the selected custom storage class is `ByteField`. Instructions for creating a user-defined custom storage class like `ByteField` appear in “Selecting and Editing CSCs, Memory Sections, and References” on page 7-15.



Properties in the **General** category, and the possible values for each property, are as follows:

- **Name:** The CSC name, selected from the **Custom storage class definitions** list. The name cannot be any TLC keyword. Violating this rule causes an error.
- **Type:** Specifies how objects of this class are stored. Values:
 - **Unstructured:** Objects of this class generate unstructured storage declarations (for example, scalar or array variables), for example:

```
datatype dataname[dimension];
```
 - **FlatStructure:** Objects of this class are stored as members of a struct. A **Structure Attributes** tab is also displayed, allowing you to specify additional properties such as the struct name. See “Structure Attributes” on page 7-23.
 - **Other:** Used for certain data layouts, such as nested structures, that cannot be generated using the standard **Unstructured** and **FlatStructure** custom storage class types. If you want to generate other types of data, you can create a new custom storage class from scratch by writing the necessary TLC code. See “Defining Advanced Custom Storage Class Types” on page 7-62 for more information.
- **For parameters** and **For signals:** These options let you enable a CSC for use with only certain classes of data objects. For example, it does not make sense to assign storage class **Const** to a **Simulink.Signal** object. Accordingly, the **For signals** option for the **Const** class is deselected, while the **For parameters** is selected.
- **Memory section:** Selects one of the memory sections defined in the **Memory Section** panel. See “Creating and Editing Memory Section Definitions” on page 7-31.
- **Data scope:** Controls the scope of symbols generated for data objects of this class. Values:
 - **Auto:** Symbol scope is determined internally by the Real-Time Workshop software. If possible, symbols have **File** scope. Otherwise, they have **Exported** scope.

- **Exported:** Symbols are exported to external code in the header file specified by the **Header File** field. If no **Header File** is specified, symbols are exported to external code in *model.h*.
- **Imported:** Symbols are imported from external code in the header file specified by the **Header File** field. If you do not specify a header file, an `extern` directive is generated in *model_private.h*. For imported data, if the **Data initialization** value is **Macro**, a header file *must* be specified.
- **File:** The scope of each symbol is the file that defines it. File scope requires each symbol to be used in a single file. If the same symbol is referenced in multiple files, an error occurs at code generation time.
- **Instance specific:** Symbol scope is defined by the **Data scope** field of the `RTWInfo.CustomAttributes` property of each data object.
- **Data initialization:** Controls how storage is initialized in generated code. Values:
 - **Auto:** Storage initialization is determined internally by the Real-Time Workshop software. Parameters have **Static** initialization, and signals have **Dynamic** initialization.
 - **None:** No initialization code is generated.
 - **Static:** A static initializer of the following form is generated:

```
datatype dataname[dimension] = {...};
```

- **Dynamic:** Variable storage is initialized at runtime, in the *model_initialize* function.
- **Macro:** A macro definition of the following form is generated:

```
#define data numeric_value
```

The **Macro** initialization option is available only for use with unstructured parameters. It is not available when the class is configured for generation of structured data, or for signals. If the **Data scope** value is **Imported**, a header file *must* be specified.

- **Instance specific:** Initialization is defined by the **Data initialization** property of each data object.

Note When necessary, the Real-Time Workshop Embedded Coder software generates dynamic initialization code for signals and states even if the CSC has **Data initialization** set to None or Static.

- **Data access:** Controls whether imported symbols are declared as variables or pointers. This field is enabled only when **Data scope** is set to Imported or Instance-specific. Values:
 - **Direct:** Symbols are declared as simple variables, such as

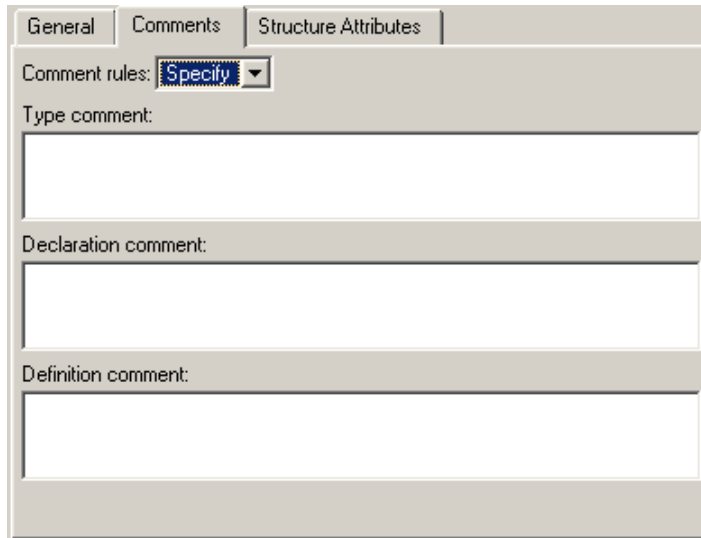
```
extern myType myVariable;
```
 - **Pointer:** Symbols are declared as pointer variables, such as

```
extern myType *myVariable;
```
 - **Instance specific:** Data access is defined by the **Data access** property of each data object.
- **Header file:** Defines the name of a header file that contains exported or imported variable declarations for objects of this class. Values:
 - **Specify:** An edit field is displayed to the right of the property. This lets you specify a header file for exported or imported storage declarations. Specify the full filename, including the filename extension (such as `.h`). Use quotes or brackets as in C code to specify the location of the header file. Leave the edit field empty to specify no header file.
 - **Instance specific:** The header file for each data object is defined by the **Header file** property of the object. Leave the property undefined to specify no header file for that object.

If the **Data scope** is Exported, specifying a header file is optional. If you specify a header file name, the custom storage class generates a header file containing the storage declarations to be exported. Otherwise, the storage declarations are exported in `model.h`.

If the **Data scope** of the class is Imported, and **Data initialization** is Macro, you *must* specify a header file name. A `#include` directive for the header file is generated.

Comments. The **Comments** panel lets you specify comments to be generated with definitions and declarations.



Comments must conform to the ANSI C standard (`/*...*/`). Use `\n` to specify a new line.

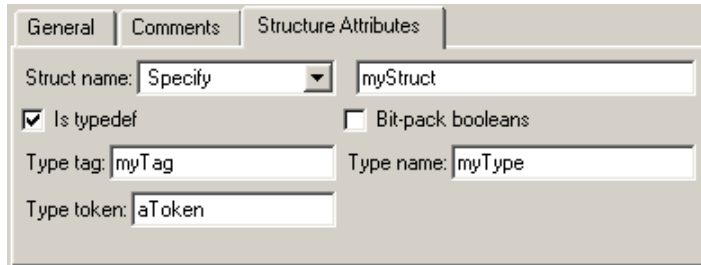
Properties in the **Comments** tab are as follows:

- **Comment rules:** If **Specify** is selected, edit fields allowing you to enter comments are displayed. If **Default** is selected, comments are generated under control of the Real-Time Workshop software.
- **Type comment:** The comment entered in this field precedes the `typedef` or `struct` definition for structured data.
- **Declaration comment:** Comment that precedes the storage declaration.
- **Definition comment:** Comment that precedes the storage definition.

Structure Attributes

The **Structure Attributes** panel gives you detailed control over code generation for structs (including bitfields). The **Structure Attributes** tab

is displayed for CSCs whose **Type** parameter is set to FlatStructure. The following figure shows the **Structure Attributes** panel.



Structure Attributes Panel

The **Structure Attributes** properties are as follows:

- **Struct name:** If you select Instance specific, specify the struct name when configuring each instance of the class.

If you select Specify, an edit field appears (as shown in Structure Attributes Panel on page 7-24) for entry of the name of the structure to be used in the struct definition. Edit fields **Type tag**, **Type token**, and **Type name** are also displayed.

- **Is typedef:** When this option is selected a typedef is generated for the struct definition, for example:

```
typedef struct {  
    ...  
} SignalDataStruct;
```

Otherwise, a simple struct definition is generated.

- **Bit-pack booleans:** When this option is selected, signals and/or parameters that have Boolean data type are packed into bit fields in the generated struct.
- **Type tag:** Specifies a tag to be generated after the struct keyword in the struct definition.
- **Type token:** Some compilers support an additional token (which is simply another string) after the type tag. To generate such a token, enter the string in this field.

- **Type name:** Specifies the string to be used in typedef definitions. This field is visible if **Is typedef** is selected.

The following listing is the pseudocode preview corresponding to the **Structure Attributes** properties displayed in Structure Attributes Panel on page 7-24.

Header file:

No header file is specified. By default, data is exported with the generated model.h file.

Type definition:

```
/* CSC type comment generated by default */  
  
typedef struct aToken myTag {  
    :  
} myType;
```

Declaration:

```
/* CSC declaration comment generated by default */  
  
extern myType MyStruct;
```

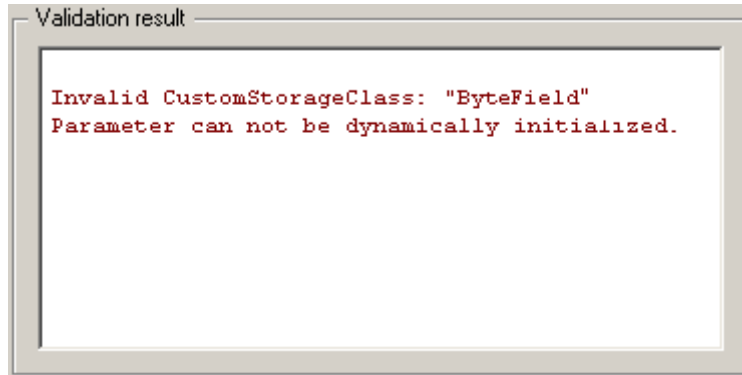
Definition:

```
/* CSC definition comment generated by default */  
  
myType MyStruct = {...};
```

Validating CSC Definitions

To validate a CSC definition, select the definition on the **Custom Storage Class** panel and click **Validate**. The Custom Storage Class Designer then checks the definition for consistency. The **Validation result** panel displays

any errors encountered when the selected CSC definition is validated. The next figure shows the **Validation result** panel with a typical error message:



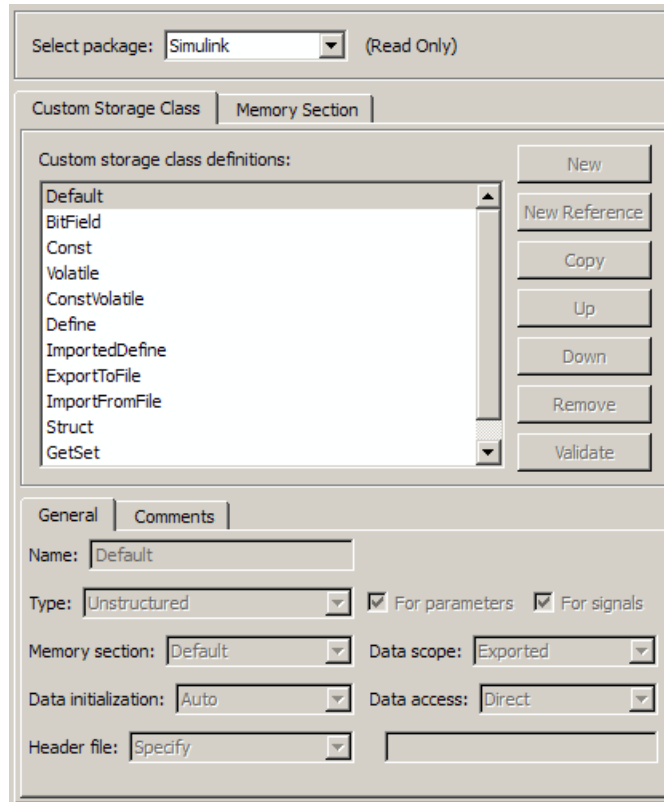
Validation is also performed whenever CSC definitions are saved. In this case, all CSC definitions are validated. (See “Saving Your Definitions” on page 7-18.)

Using Custom Storage Class References

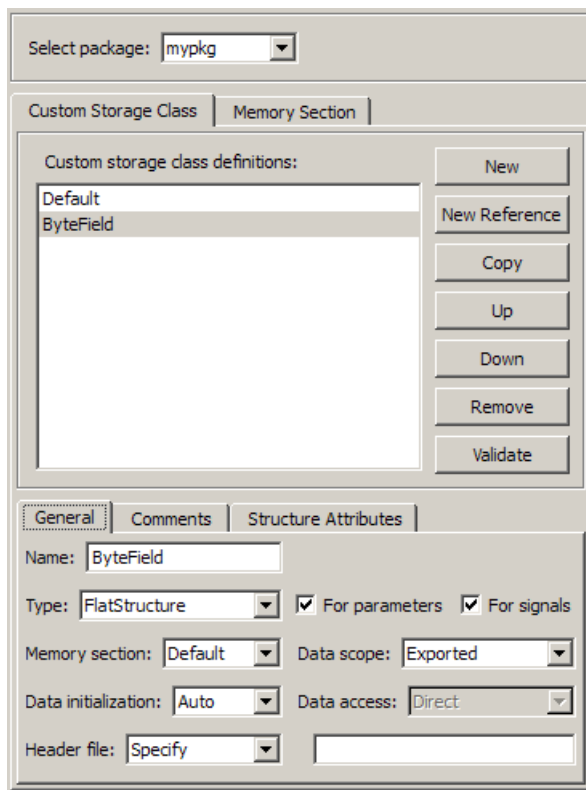
Any package can access and use custom storage classes that are defined in any other package, including both user-defined packages and predefined packages such as Simulink and mpt. Only one copy of the storage class exists, in the package that first defined it. Other packages refer to it by pointing to it in its original location. Thus any changes to the class, including changes to a predefined class in later MathWorks product releases, are immediately available in every referencing package.

To configure a package to use a custom storage class that is defined in another package:

- 1 Type `cscdesigner` to launch the Custom Storage Class Designer. The relevant part of the designer window initially looks like this:

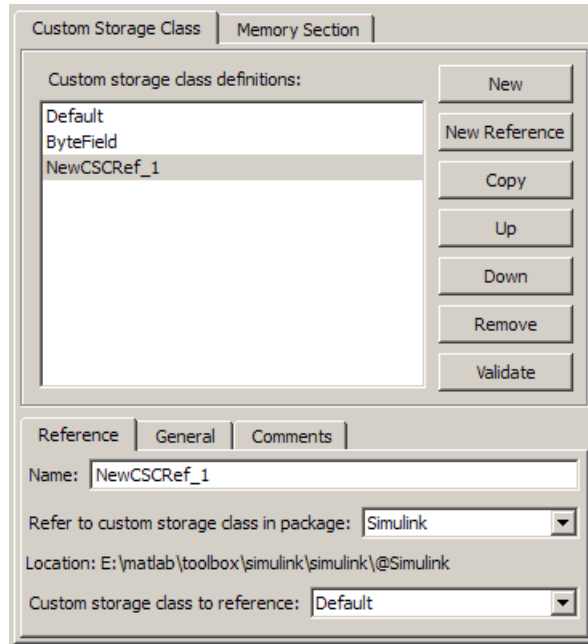


- 2** Select the **Custom Storage Class** tab.
- 3** Use **Select Package** to select the package in which you want to reference a class or section defined in some other package. The selected package must be writable.
- 4** In the **Custom storage class definitions** pane, select the existing definition below which you want to insert the reference. For example:



5 Click **New Reference**.

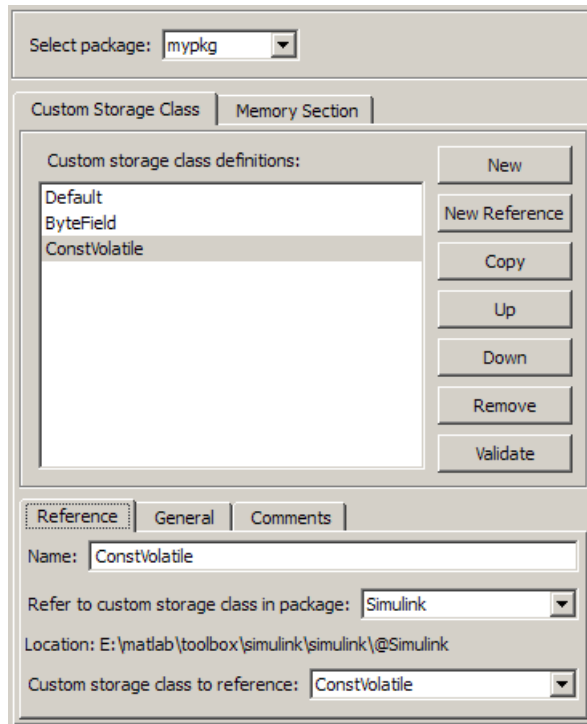
A new reference with a default name and properties appears below the previously selected definition. The new reference is selected, and a **Reference** tab appears that shows the reference's initial properties. A typical appearance is:



- 6** Use the **Name** field to enter a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package. The name cannot be any TLC keyword. Violating this rule causes an error.
- 7** Set **Refer to custom storage class in package** to specify the package that contains the custom storage class you want to reference.
- 8** Set **Custom storage class to reference** to specify the custom storage class to be referenced. Trying to create a circular reference generates an error and leaves the package unchanged.
- 9** Click **OK** or **Apply** to save the changes to memory. See “Saving Your Definitions” on page 7-18 for information about saving changes permanently.

For example, the next figure shows the custom storage class `ConstVolatile` imported from the `Simulink` package into `mypkg`, and given the same name

that it has in the source package. Any other name could have been used without affecting the properties of the storage class.



You can use Custom Storage Class Designer capabilities to copy, reorder, validate, and otherwise manage classes that have been added to a class by reference. However, you cannot change the underlying definitions. You can change a custom storage class only in the package where it was originally defined.

Changing Existing CSC References

To change an existing CSC reference, select it in the **Custom storage class definitions** pane. The **Reference** tab appears, showing the current properties of the reference. Make any needed changes, then click **OK** or **Apply** to save the changes to memory. See “Saving Your Definitions” on page 7-18 for information about saving changes permanently.

Creating and Editing Memory Section Definitions

Memory section definitions add comments, qualifiers, and #pragma directives to generated symbol declarations. The **Memory Section** tab lets you create, view, edit, and verify memory section definitions. The steps for creating a memory section definition are essentially the same as for creating a custom storage class definition:

- 1** Select a writable package in the **Select package** field.
- 2** Select the **Memory Section** tab. In a new package, only a Default memory section initially appears.
- 3** Select the existing memory section below which you want to create a new memory section.
- 4** Click **New**.

A new memory section definition with a default name appears below the selected memory section.

- 5** Set the name and other properties of the memory section as needed.
- 6** Click **OK** or **Apply**.

The next figure shows mypkg with a memory section called MyMemSect:

The screenshot shows the 'Memory Section' tab of the 'Custom Storage Class' dialog. At the top, there is a 'Select package:' dropdown menu with 'mypkg' selected. Below this, the 'Memory section definitions:' list contains two entries: 'Default' and 'MyMemSect'. To the right of this list are several buttons: 'New', 'New Reference', 'Copy', 'Up', 'Down', 'Remove', and 'Validate'. The 'Memory Section' sub-tab is active, showing the following fields: 'Name:' with 'MyMemSect' entered; 'Is const' and 'Is volatile' checkboxes, both checked, followed by a 'Qualifier:' text box; a 'Comment:' text area; 'Pragma surrounds:' dropdown menu with 'All variabl' selected; 'Pre-memory-section pragma:' text area; and 'Post-memory-section pragma:' text area.

The **Memory section definitions** list lets you select a memory section definition to view or edit. The available memory section definitions also appear in the **Memory section name** menu in the **Custom Storage Class** panel. The properties of a memory section definition are as follows:

- **Memory section name:** Name of the memory section (displayed in **Memory section definitions** list).

- **Is const:** If selected, a `const` qualifier is added to the symbol declarations.
- **Is volatile:** If selected, a `volatile` qualifier is added to the symbol declarations.
- **Qualifier:** The string entered into this field is added to the symbol declarations as a further qualifier. Note that no verification is performed on this qualifier.
- **Memory section comment:** Comment inserted before declarations belonging to this memory section. Comments must conform to the ANSI C standard (`/*...*/`). Use `\n` to specify a new line.
- **Pragma surrounds:** Specifies whether the pragma should surround `All variables` or `Each variable`. When **Pragma surrounds** is set to `Each variable`, the `%<identifier>` token is allowed in pragmas and will be replaced by the variable or function name.
- **Pre-memory section pragma:** pragma directive that precedes the storage definition of data belonging to this memory section. The directive must begin with `#pragma`.
- **Post-memory section pragma:** pragma directive that follows the storage definition of data belonging to this memory section. The directive must begin with `#pragma`.

Previewing Generated Code

If you click **Validate** on the **Memory Section** panel, the **Pseudocode preview** panel displays a preview of code that is generated from objects of the given class. The panel also displays messages (in blue) to highlight changes as they are made. The code preview changes dynamically as you edit the class properties. The next figure shows a code preview for the `MemConstVolatile` memory section.

```
Pseudocode preview
-----
Header file: Not applicable.
-----
Type definition: Not applicable.
-----
Declaration:
extern const volatile DATATYPE DATANAME;
-----
Definition:
/* ConstVolatile memory section */
const volatile DATATYPE DATANAME;
```

Using Memory Section References

Any package can access and use memory sections that are defined in any other package, including both user-defined packages and predefined packages such as Simulink and mpt. Only one copy of the section exists, in the package that first defined it; other packages refer to it by pointing to it in its original location. Thus any changes to the section, including changes to a predefined section in later MathWorks product releases, are immediately available in every referencing package.

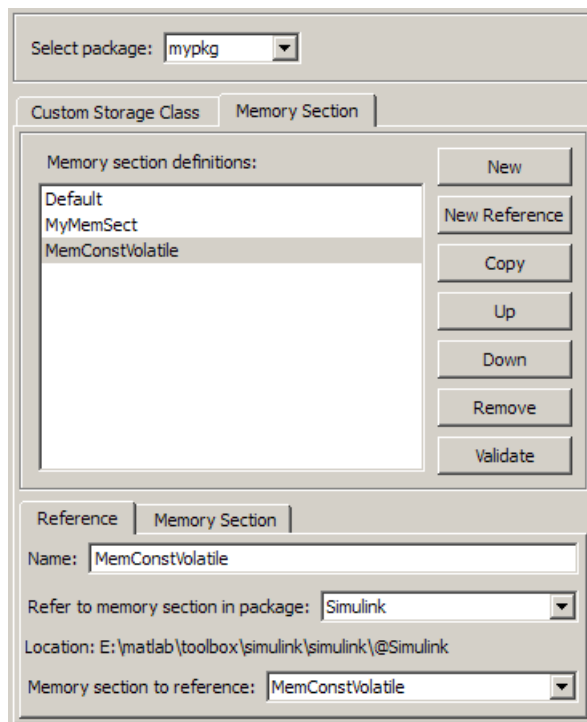
To configure a package to use a memory section that is defined in another package:

- 1 Type `cscdesigner` to launch the Custom Storage Class Designer.
- 2 Select the **Memory Section** tab.
- 3 Use **Select Package** to select the package in which you want to reference a class or section defined in some other package.
- 4 In the **Memory section definitions** pane, select the existing definition below which you want to insert the reference.
- 5 Click **New Reference**.

A new reference with a default name and properties appears below the previously selected definition. The new reference is selected, and a **Reference** tab appears that shows the reference's initial properties.

- 6** Use the **Name** field to enter a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package.
- 7** Set **Refer to memory section in package** to specify the package that contains the memory section you want to reference.
- 8** Set **Memory section to reference** to specify the memory section to be referenced. Trying to create a circular reference generates an error and leaves the package unchanged.
- 9** Click **OK** or **Apply** to save the changes to memory. See “Saving Your Definitions” on page 7-18 for information about saving changes permanently.

For example, the next figure shows the memory section `MemConstVolatile` imported from the `Simulink` package into `mypkg`, and given the same name that it has in the source package. Any other name could have been used without affecting the properties of the memory section.



You can use Custom Storage Class Designer capabilities to copy, reorder, validate, and otherwise manage memory sections that have been added to a class by reference. However, you cannot change the underlying definitions. You can change a memory section only in the package where it was originally defined.

Changing Existing Memory Section References

To change an existing memory section reference, select it in the **Memory section definitions** pane. The **Reference** tab appears, showing the current properties of the reference. Make any needed changes, then click **OK** or **Apply** to save the changes to memory. See “Saving Your Definitions” on page 7-18 for information about saving changes permanently.

Applying CSCs to Parameters and Signals

In this section...

- “About Applying Custom Storage Classes” on page 7-37
- “Applying a Custom Storage Class to a Parameter” on page 7-38
- “Applying a Custom Storage Class to a Signal” on page 7-40
- “Applying a CSC Using a Base Workspace Signal Object” on page 7-41
- “Applying a CSC Using an Embedded Signal Object” on page 7-43
- “Specifying a Custom Storage Class Using the GUI” on page 7-50
- “Specifying a Custom Storage Class Using the API” on page 7-53

About Applying Custom Storage Classes

You can apply a custom storage class to a parameter or a signal using the GUI or the API.

- To apply a custom storage class to a parameter, you specify the storage class in the `Simulink.Parameter` object that defines the parameter in the base workspace.
- To apply a custom storage class to a signal, you specify the storage class in a `Simulink.Signal` object that is bound to the signal. You can provide this object in two ways:
 - Create the object in the base workspace, then bind it to the signal as described in “Resolving Symbols”. When you save the model, you must save the object in a separate file, as with any base workspace object.
 - Use the Signal Properties dialog box to embed the object in the model on the port where the signal originates. When you save the model, Simulink automatically saves the embedded signal object as part of the model file.

Most of the GUI techniques, and most of the API techniques, are the same for parameter and signal objects, and for base workspace and embedded signal objects. Only the initial steps differ, after which you apply the same GUI or API instructions within the context that you established in the initial steps.


The following instructions assume that you have already created any needed packages, custom storage classes, and memory sections, as described in “Creating Packages that Support CSC Definitions” on page 7-8 and “Designing Custom Storage Classes and Memory Sections” on page 7-12.

Applying a Custom Storage Class to a Parameter

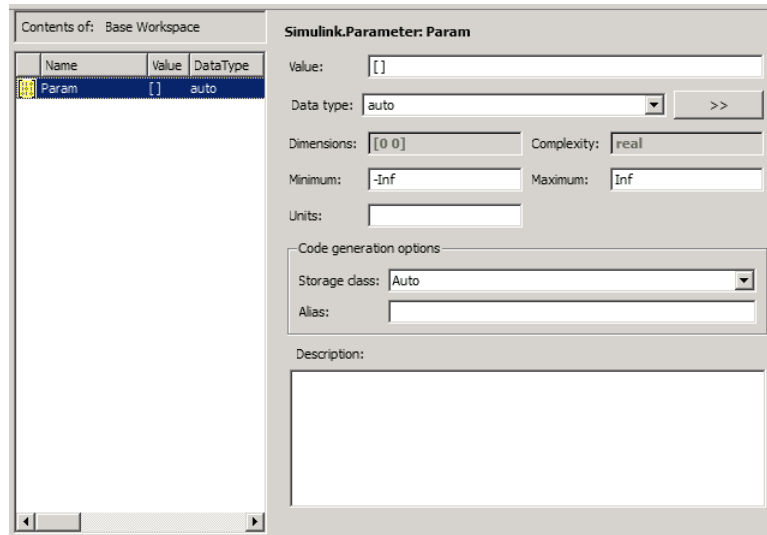
To apply a custom storage class to a parameter, you specify the storage class in the `Simulink.Parameter` object that defines the parameter in the base workspace. The instructions that begin in this section show you how to create that object using the GUI or API. Later instructions show you how to specify a custom storage class and custom attributes.

For information about using parameter objects to specify block parameter values, see “Working with Block Parameters” in the Simulink documentation. For information about parameter storage in generated code, see “Parameter Considerations” in the Real-Time Workshop documentation.

Providing a Parameter Object Using the GUI

- 1 In the Model window, choose **View > Model Explorer**.
- 2 In the **Model Hierarchy** pane, select the Base Workspace.
- 3 Click the **Add Parameter** tool  or choose **Add > Simulink Parameter**.

Simulink creates a `Simulink.Parameter` object with a default name:



- 4 Change the parameter name as needed by editing it in the **Contents** pane. Example: MyParam.
- 5 Set parameter attributes other than **Code generation options** in the **Dialog** pane.
- 6 Follow the instructions in “Specifying a Custom Storage Class Using the GUI” on page 7-50.

Providing a Parameter Object Using the API

- 1 In the MATLAB Command Window, enter:

```
ParamName=ParamClass
```

where *ParamClass* is `Simulink.Parameter` or any subclass of it that you have defined.

- 2 Simulink creates a *ParamClass* object with the specified name:

```
MyParam =  
  
Simulink.Parameter (handle)
```

```
Value: []
RTWInfo: [1x1 Simulink.ParamRTWInfo]
Description: ''
DataType: 'auto'
Min: -Inf
Max: Inf
DocUnits: ''
Complexity: 'real'
Dimensions: [0 0]
```

- 3 Set parameter attributes other than `RTWInfo`, which controls custom storage classes.
- 4 Follow the instructions in “Specifying a Custom Storage Class Using the API” on page 7-53.

Applying a Custom Storage Class to a Signal

To apply a custom storage class to a signal, you specify the storage class in a `Simulink.Signal` object. This object can exist in either of two locations:

- In the MATLAB base workspace
- On the port where the signal originates

The object itself is the same in either case; only its location and some of the techniques for managing it differ. The instructions that begin in this section show you how to create a signal object in either location using the GUI or API. Later instructions show you how to specify the custom storage class and custom attributes.

A given signal can be associated with at most one signal object under any circumstances. The signal can refer to the object more than once, but every reference must resolve to exactly the same object. A different signal object that has exactly the same properties will not meet the requirement for uniqueness. A compile-time error occurs if a model associates more than one signal object with any signal.

Assigning a signal to any non-Auto storage class automatically makes the signal a test point, overriding the setting of **Signal Properties > Logging**


and accessibility > Test point. See “Working with Test Points” for more information.

For information about using signal objects to specify signal attributes, see “Working with Signals” in the Simulink documentation. For information about signal storage in generated code, see “Signal Considerations” in the Real-Time Workshop documentation.

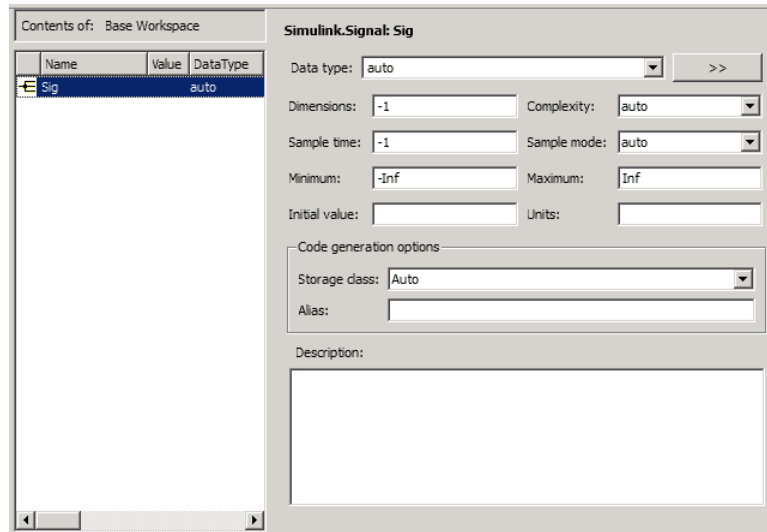
Applying a CSC Using a Base Workspace Signal Object

The first step is to create the signal object in the base workspace, after which you specify any needed signal attributes and the custom storage class and attributes.

Providing a Base Workspace Signal Object Using the GUI

- 1 In the Model window, choose **View > Model Explorer**.
- 2 In the **Model Hierarchy** pane, select the Base Workspace.
- 3 Click the **Add Signal** tool  or choose **Add > Simulink Signal**.

Simulink creates a `Simulink.Signal` object with a default name:



- 4 Change the signal name as needed by editing it in the **Contents** pane.
Example: MySig.
- 5 Set signal attributes other than **Code generation options** in the **Dialog** pane.
- 6 Give the signal the same name as the signal object, as described in “Naming Signals”.
- 7 Arrange for the signal to resolve to the object, as described in “Resolving Symbols”.
- 8 Follow the instructions in “Specifying a Custom Storage Class Using the GUI” on page 7-50.

Providing a Base Workspace Signal Object Using the API

- 1 In the MATLAB Command Window, enter:

```
SignalName=SignalClass
```

where *SignalClass* is `Simulink.Signal` or any subclass of it that you have defined.

- 2** Simulink creates a *SignalClass* object with the specified name:

```
MySig =

Simulink.Signal (handle)
    RTWInfo: [1x1 Simulink.SignalRTWInfo]
    Description: ''
    DataType: 'auto'
    Min: -Inf
    Max: Inf
    DocUnits: ''
    Dimensions: -1
    Complexity: 'auto'
    SampleTime: -1
    SamplingMode: 'auto'
    InitialValue: ''
```

- 3** Set parameter attributes other than *RTWInfo*, which controls custom storage classes.
- 4** Give the signal the same name as the signal object, as described in “Naming Signals”.
- 5** Arrange for the signal to resolve to the object, as described in “Resolving Symbols”.
- 6** Follow the instructions in “Specifying a Custom Storage Class Using the API” on page 7-53.

Applying a CSC Using an Embedded Signal Object

You can use the GUI or the API to apply a CSC using an embedded signal object.

- If you use the GUI, you use the Signal Properties dialog box to specify the attributes you want. The software then creates a *Simulink.Signal* object and assigns it to the output port where the signal originates.
- If you use the API, you instantiate *Simulink.Signal* or a subclass of it, set the attribute values that you want, and assign the object to the output port where the signal originates.

In either case, the effect on code generation is the same as if you had created a base workspace signal object that specified the same name, CSC, and custom attributes as the embedded signal object. See “Applying a CSC Using a Base Workspace Signal Object” on page 7-41 for details.

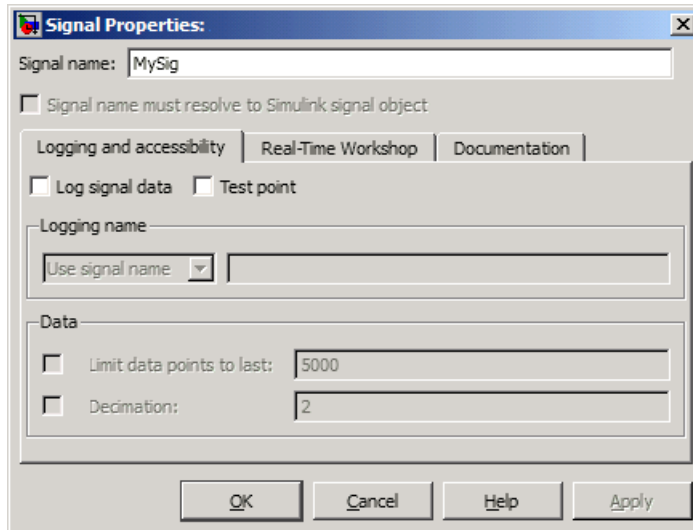
The advantages of using embedded signal objects are that they do not clutter the base workspace, and they do not need to be saved separately from the model, as base workspace objects do. When you save a model, Simulink saves any embedded signal objects in the model file, and reloads the objects when you later reload the model.

The disadvantage of embedded signal objects is that you can use such an object *only* to specify a custom storage class, custom attributes, and an alias; you must accept the default values for all other signal attributes. You cannot work around this restriction by providing additional information in a base workspace signal object on the same signal, because a signal object can have at most one associated signal object, as described in “Multiple Signal Objects”.

Providing an Embedded Signal Object using the GUI

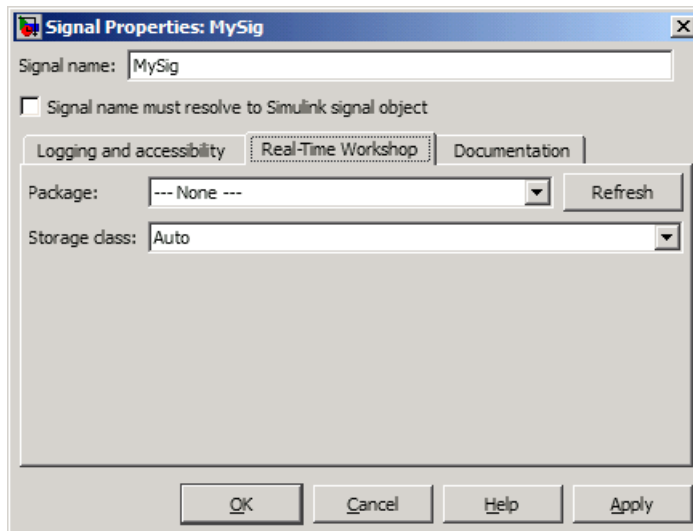
- 1 Give the signal a name, which must be a valid ANSI C identifier. Example: MySig.
- 2 Right-click the signal and choose **Signal Properties** from the context menu.

The Signal Properties dialog box opens:

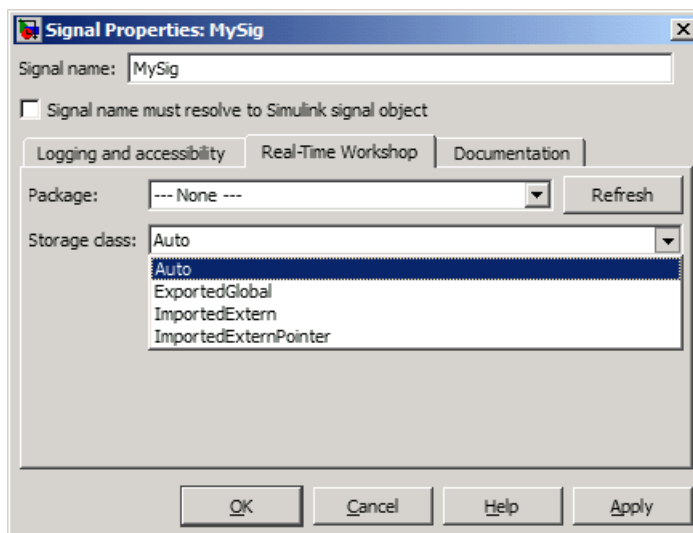


3 *Do not* select **Signal name must resolve to Simulink signal object**. Selecting it would require a base workspace signal object, which would conflict with the embedded signal object.

4 Click the **Real-Time Workshop** tab:

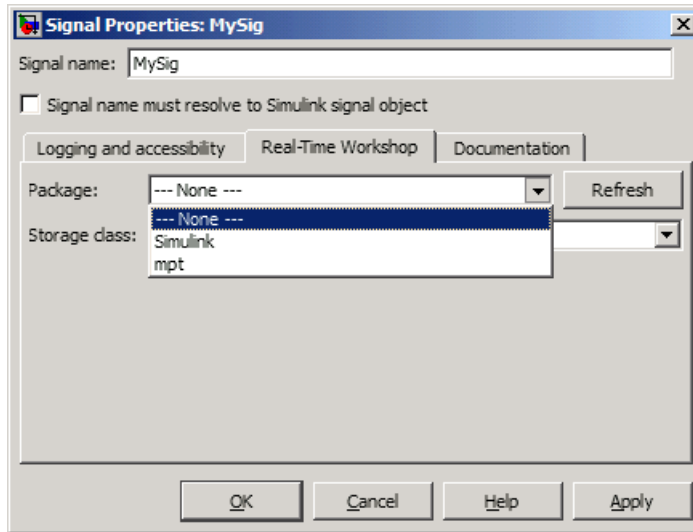


- 5 The **Package** is initially ---None---. When no package is specified, only the non-custom built-in storage classes defined for both GRT and ERT targets are available:

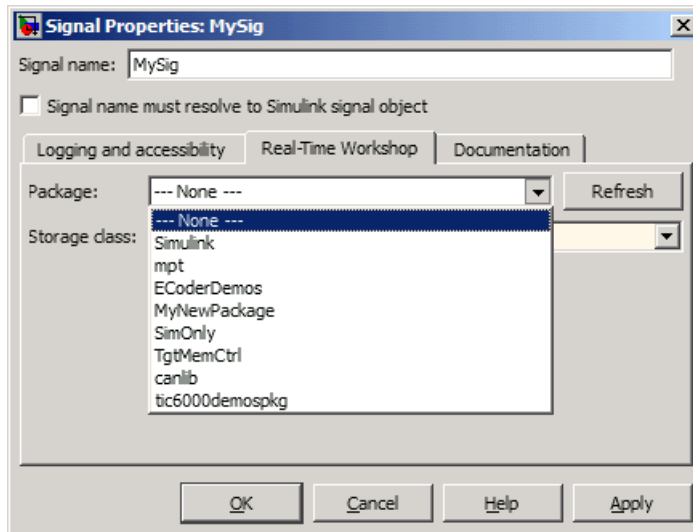


Applying a storage class when the package is ---None--- sets internal storage class attributes rather than creating an embedded signal object. For more information, see “Signal Considerations” and “Simulink Data Object Considerations” in the Real-Time Workshop documentation.

- 6 To apply a custom storage class, you must first specify the package where it is defined. Initially, viewing the **Package** menu displays only the built-in Simulink and mpt packages:

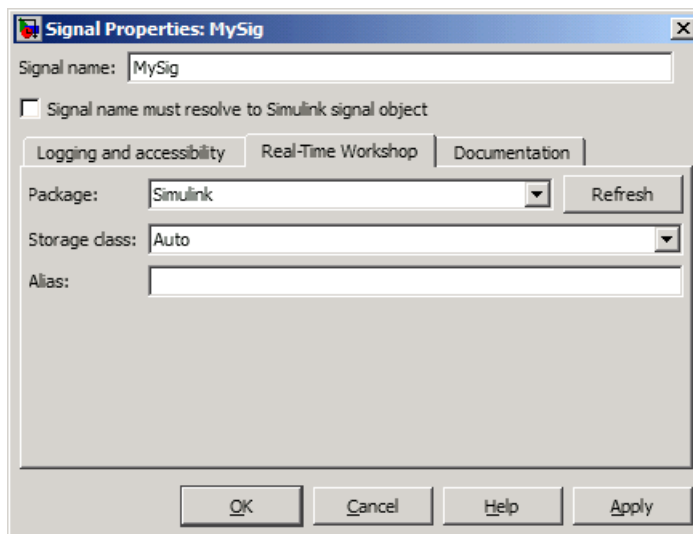


- 7 Click **Refresh** to load any other available packages, including user-defined packages, available on the MATLAB path. After a brief delay, a timer box tracks the progress of the package search. After the search completes, viewing the **Package** menu displays all available packages:



Once you have used **Refresh** in any Signal Properties dialog, Simulink saves the information for later use, so you do not have to click **Refresh** again during the current MATLAB session.

- 8 Select the package that contains the custom storage class you want to apply, e.g. Simulink:



- 9 Follow the instructions in “Specifying a Custom Storage Class Using the GUI” on page 7-50.

Deleting an Embedded Signal Object Using the GUI

To delete an embedded signal object with the GUI, delete the name of the signal to which the object applies, by editing the name in the graphical model or in the Signal Properties dialog box. Simulink automatically deletes the embedded signal object as soon as its signal has no name.

Providing an Embedded Signal Object using the API

To provide an embedded signal object using the API, you create the object, set its custom storage class and any custom attributes, then assign the object to the output port on which it will be embedded.

- 1 Name the signal if it does not already have a name. The name must be a valid ANSI C identifier.
- 2 In the MATLAB Command Window, enter:

```
SignalName=SignalClass
```

where *SignalClass* is `Simulink.Signal` or any subclass of it that you have defined. The name of the signal object does not need to match the name of the signal to which the object will be applied.

- 3 Simulink creates a *SignalClass* object with the specified name. Example:

```
MySig =

Simulink.Signal (handle)
  RTWInfo: [1x1 Simulink.SignalRTWInfo]
  Description: ''
  DataType: 'auto'
  Min: -Inf
  Max: Inf
  DocUnits: ''
  Dimensions: -1
  Complexity: 'auto'
  SampleTime: -1
  SamplingMode: 'auto'
  InitialValue: ''
```

- 4 *Do not* set any attributes. An embedded signal object can specify *only* custom storage class information.
- 5 Follow the instructions in “Specifying a Custom Storage Class Using the API” on page 7-53. After specifying the custom storage class, be sure to assign the signal object to its output port, as described under “Assigning an Embedded Signal Object to an Output Port” on page 7-57.

Changing an Embedded Signal Object Using the API

To change an embedded signal object using the API, you obtain a copy of the object from the output port on which it is embedded, change the object as needed, then assign the changed object back to the port.

- 1 Obtain a copy of the signal object using a handle to the output port.
Example:

```
hps=get_param(gcb, 'PortHandles')
hp=hps.Outputport(1)
MySig=get_param(hp, 'SignalObject')
```

- 2 Change the signal object using the techniques described in “Specifying a Custom Storage Class Using the API” on page 7-53. After making the changes, be sure to copy the signal object to its output port, as described in “Assigning an Embedded Signal Object to an Output Port” on page 7-57.

Deleting an Embedded Signal Object Using the API

To delete an embedded signal object with the API, obtain a handle to the output port where the signal object is embedded, then set the port’s `SignalObject` parameter to []:

```
hps=get_param(gcb, 'PortHandles')
hp=hps.Outputport(1)
set_param(hp, 'SignalObject', [])
```

Specifying a Custom Storage Class Using the GUI

The initial steps for applying a CSC with the GUI differ depending on whether you are applying the CSC to a parameter using a base workspace object, to a signal using a base workspace object, or to a signal using an embedded object. The initial steps for each of these three cases appear in:

- “Providing a Parameter Object Using the GUI” on page 7-38
- “Providing a Base Workspace Signal Object Using the GUI” on page 7-41
- “Providing an Embedded Signal Object using the GUI” on page 7-44

After the initial steps, applying a CSC with the GUI is the same in all three cases. The following instructions show you how to finish applying a CSC with the GUI. The instructions assume that you have completed one of the previous sets of instructions, and that the dialog you used to execute those instructions is still open. If necessary, return to the relevant section and restore the situation that existed at its end, then return to this section.

The instructions given in this section apply to all packages, but the available custom storage classes and custom attributes depend on the package that you select. The examples in this section assume that you are using the Simulink package.

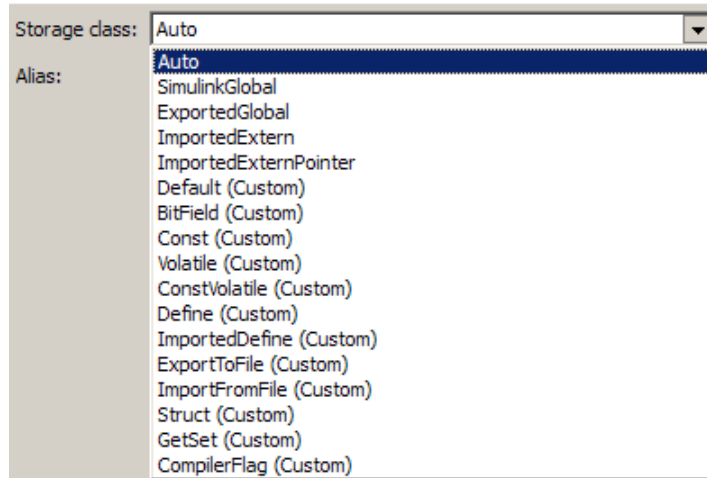
The dialog that you used to begin the process of applying a CSC with the GUI by providing an object contains two fields: one for specifying a custom storage class and one for optionally specifying an alias. These fields are the same in all three of the dialogs that you might use:

A screenshot of a dialog box with a light gray background. It contains two input fields. The first field is labeled "Storage class:" and has a dropdown menu with "Auto" selected. The second field is labeled "Alias:" and is currently empty.

Storage class is Auto because that is the default storage class in the Simulink package. Other packages may have different defaults. You can specify an **Alias** whenever the **Storage class** is not Auto. If **Storage class** is Auto, Simulink deletes any alias you try to specify, leaving the field blank. If you specify an alias, it appears in generated code instead of the name of the object.

To specify a custom storage class and its custom attributes:

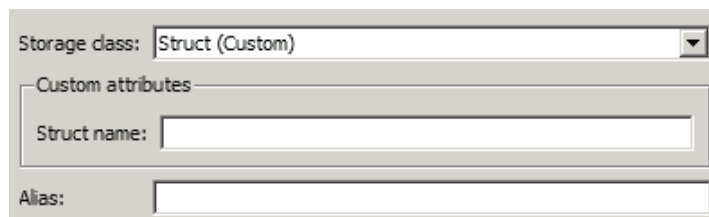
- 1 View the **Storage Class** menu, which looks like this for the Simulink package:



Each custom storage class has (custom) suffixed to its name. The storage classes `SimulinkGlobal`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer` are the built-in non-custom storage classes described in “Signal Considerations” and “Simulink Data Object Considerations” in the Real-Time Workshop documentation.

- 2 Choose the desired custom storage class from **Storage class**, for example, **Struct**.

If the storage class defines any custom attributes, fields for defining them appear:



- 3 Provide values for any custom attributes. **Struct** has only one, **Struct name**. For example, set **Struct name** to `MyStruct`:

The image shows a configuration window with the following fields:

- Storage class:** A dropdown menu currently showing "Struct (Custom)".
- Custom attributes:** A section containing:
 - Struct name:** A text input field containing "MyStruct".
 - Alias:** An empty text input field.

4 Click **Apply**.

In generated code, all data whose storage is controlled by this custom storage class specification will appear in a structure named `MyStruct`. See “Generating Code with Custom Storage Classes” on page 7-58 for an example.

Specifying a Custom Storage Class Using the API

The initial steps for applying a CSC with the API differ depending on whether you are applying the CSC to a parameter using a base workspace object, to a signal using a base workspace object, or to a signal using an embedded object. The initial steps for each of these three cases appear in:

- “Providing a Parameter Object Using the API” on page 7-39
- “Providing a Base Workspace Signal Object Using the API” on page 7-42
- “Providing an Embedded Signal Object using the API” on page 7-48

After the initial steps, applying a CSC with the API is the same in all three cases, except for an assignment needed only by an embedded signal object. The following instructions show you how to finish applying a CSC with the API. The instructions assume that you have completed one of the previous sets of instructions, and that the resulting objects and attributes are unchanged. If necessary, return to the relevant section and restore the situation that existed at its end, then return to this section.

The instructions given in this section apply to all packages, but the available custom storage classes and custom attributes depend on the package that you select. The examples in this section assume that you are using the `Simulink` package. The examples also assume that the object for which you want to specify a custom storage class is named `MyObj`, which is a parameter or

signal object that exists in the base workspace, or a signal object that will be assigned to an output port.

The rest of this section provides information that is specific to custom storage classes in Real-Time Workshop Embedded Coder. See “Simulink Package Custom Storage Classes” on page 7-6 for a list of the custom storage classes that are built into the Simulink package for use by Real-Time Workshop Embedded Coder software.

RTWInfo Properties

Each Simulink parameter object or signal object defines properties called RTWInfo properties. Real-Time Workshop uses these properties to control storage class assignment in generated code. The RTWInfo properties and their default values are:

```
StorageClass: 'Auto'  
Alias: ''  
CustomStorageClass: 'Default'  
CustomAttributes: [1x1 SimulinkCSC.AttribClass_Simulink_Default]
```

For more information about RTWInfo properties, see “Signal Considerations” and “Simulink Data Object Considerations” in the Real-Time Workshop documentation.

Specifying a Custom Storage Class

To specify a custom storage class using RTWInfo properties:

- 1 Set `StorageClass` to `'Custom'`.
- 2 Set `CustomStorageClass` to the name of the storage class.

For example, to specify the Struct custom storage class:

```
MyObj.RTWInfo.StorageClass='Custom'  
MyObj.RTWInfo.CustomStorageClass='Struct'
```

Whenever you have specified a custom storage class other than `Auto`, you can specify an alias by setting the `Alias` attribute. If you specify an alias, it appears in generated code instead of the name of the object.

Specifying Instance-Specific Attributes

A custom storage class can have properties that define attributes that are specific to that CSC. Such properties are called **instance-specific attributes**. For example, if you specify the `Struct` custom storage class, you must specify the name of the C language structure that will store the data. That name is an instance-specific attribute of the `Struct` CSC.

Instance-specific attributes are stored in the `RTWInfo.CustomAttributes` property. This property is initially defined as follows:

```
SimulinkCSC.AttribClass_Simulink_Default
1x1 struct array with no fields
```

When you specify a custom storage class, Simulink automatically populates `RTWInfo.CustomAttributes` with the fields necessary to represent any instance-specific attributes of that CSC. For example, if you set the `MySig` CSC to `Struct`, as described in “Specifying a Custom Storage Class” on page 7-54, then enter:

```
MyObj.RTWInfo.CustomAttributes
```

MATLAB displays:

```
SimulinkCSC.AttribClass_Simulink_Struct
StructName: ''
```

To specify that `StructName` is `MyStruct`, enter:

```
MyObj.RTWInfo.CustomAttributes.StructName='MyStruct'
```

MATLAB displays:

```
SimulinkCSC.AttribClass_Simulink_Struct
StructName: 'MyStruct'
```

Simulink Package CSC Instance-Specific Properties

Class Name	Instance-Specific Property	Purpose
BitField	CustomAttributes.StructName	Name of the bitfield struct into which the code generator packs the object's Boolean data.
ExportToFile	CustomAttributes.HeaderFile	Name of header (.h) file that contains exported variable declarations and export directives for the object.
GetSet	CustomAttributes.HeaderFile	Name of header (.h) file to #include in the generated code. See "GetSet Custom Storage Class for Data Store Memory" on page 7-66.
	CustomAttributes.GetFunction	String that specifies the name of a function call to read data.
	CustomAttributes.SetFunction	String that specifies the name of a function call to write data.
ImportedDefine	CustomAttributes.HeaderFile	The header file that defines the values of code variant preprocessor conditionals. See "Generating Code Variants for Variant Models" on page 7-22.
ImportFromFile	CustomAttributes.HeaderFile	Name of header (.h) file containing global variable declarations the code generator imports for the object.
Struct	CustomAttributes.StructName	Name of the struct into which the code generator packs the object's data.

Assigning an Embedded Signal Object to an Output Port

If you are operating on an embedded signal object with the API, you must copy the object to the port after providing or changing its `RTWInfo` properties. For example, if `MyObj` is a signal object that you want to copy to the output port, enter:

```
hps=get_param(gcb, 'PortHandles')
hp=hps.Output(1)
set_param(hp, 'SignalObject', 'MyObj')
```

Subsequent changes to the source object in the base workspace have no effect on the output port copy, and you can delete the source object if you have no further use for it:

```
clear ('MyObj')
```

Generating Code with Custom Storage Classes

In this section...

“Code Generation Prerequisites” on page 7-58

“Code Generation Example” on page 7-58

Code Generation Prerequisites

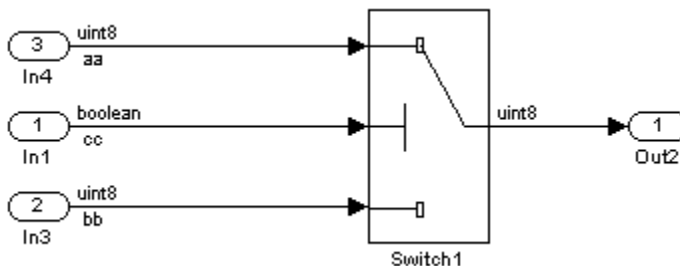
Before you generate code for a model that uses custom storage classes, set model options as follows:

- If your model assigns custom storage classes to any parameters, select **Configuration Parameters > Optimization > Simulation and code generation > Inline parameters**. Otherwise, the code generator ignores CSC specifications for parameters. This requirement also applies to models that assign built-in storage classes to parameters.
- Clear **Configuration Parameters > Real-Time Workshop > Data specification override > Ignore custom storage classes**.

Otherwise, the code generator ignores all CSC specifications, and treats all data objects as if their **Storage class** were Auto.

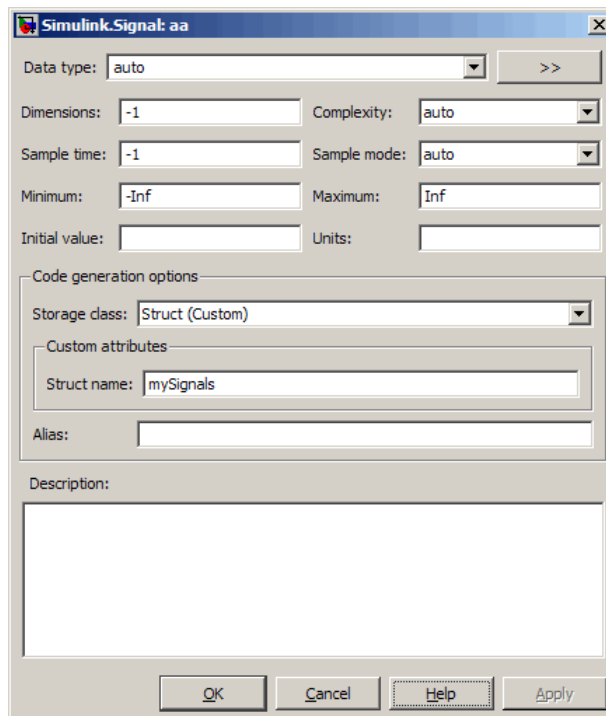
Code Generation Example

This section presents an example of code generation with CSCs, based on this model:



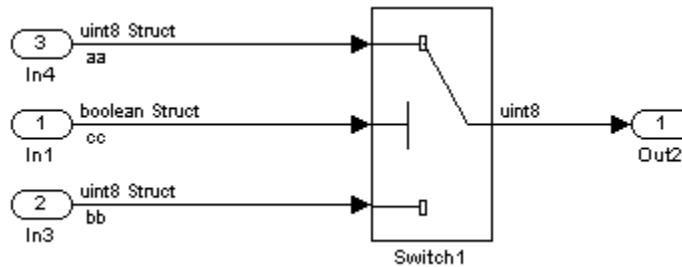
The model contains three named signals: `aa`, `bb`, and `cc`. Using the predefined `Struct` custom storage class, the example generates code that packs these signals into a `struct` named `mySignals`. The `struct` declaration is then exported to externally written code.

To specify the `struct`, you provide `Simulink.Signal` objects that specify the `Struct` custom storage class, and associate the objects with the signals as described in “Applying CSCs to Parameters and Signals” on page 7-37. All three objects have the same properties. This figure shows the signal object properties for `aa`:



The association between identically named model signals and signal objects is formed as described in “Resolving Symbols”. In this example, the symbols `aa`, `bb`, and `cc` resolve to the signal objects `aa`, `bb`, and `cc`, which have custom storage class `Struct`. In the generated code, storage for the three signals will be allocated within a `struct` named `mySignals`.

You can display the storage class of the signals in the block diagram by selecting **Port/Signal Display > Storage Class** from the Simulink model editor **Format** menu. The figure below shows the block diagram with signal data types and signal storage classes displayed.



With the model configured as described in “Code Generation Prerequisites” on page 7-58, and the signal objects defined and associated with the signals, you can generate code that uses the custom storage classes to generate the desired data structure for the signals. After code generation, the relevant definitions and declarations are located in three files:

- *model_types.h* defines the following struct type for storage of the three signals:

```
typedef struct MySignals_tag {
    boolean_T cc;
    uint8_T bb;
    uint8_T aa;
} mySignals_type;
```

- *model.c* (or *.cpp*) defines the variable `mySignals`, as specified in the object’s instance-specific `StructName` attribute. The variable is referenced in the code generated for the Switch block:

```
/* Definition for Custom Storage Class: Struct */

mySignals_type mySignals = {
    /* cc */
    FALSE,
    /* bb */
```

```

0,
/* aa */
0
};
...
/* Switch: '<Root>/Switch1' */
if(mySignals.cc) {
    rtb_Switch1 = mySignals.aa;
} else {
    rtb_Switch1 = mySignals.bb;
}

```

- `model.h` exports the `mySignals` Struct variable:

```

/* Declaration for Custom Storage Class: Struct */

extern mySignals_type mySignals;

```

Grouped Custom Storage Classes

A custom storage class that results in multiple data objects being referenced with a single variable in the generated code, in the previous example, is called a *grouped custom storage class*. In the Simulink package, `Bitfield` and `Struct` (shown in the preceding example) are grouped CSCs. Data grouped by a CSC is referred to as *grouped data*.

Defining Advanced Custom Storage Class Types

In this section...
“Introduction” on page 7-62
“Create Your Own Parameter and Signal Classes” on page 7-62
“Create a Custom Attributes Class for Your CSC (Optional)” on page 7-62
“Write TLC Code for Your CSC” on page 7-63
“Register Custom Storage Class Definitions” on page 7-63

Introduction

Certain data layouts, such as nested structures, cannot be generated using the standard `Unstructured` and `FlatStructure` custom storage class types. You can define an *advanced custom storage class* if you want to generate other types of data. Creating advanced CSCs requires understanding TLC programming and using a special advanced mode of the Custom Storage Class Designer. This sections explain how to define advanced CSC types.

Create Your Own Parameter and Signal Classes

The first step is to use the Simulink Data Class Designer to create your own package containing classes derived from `Simulink.Parameter` or `Simulink.Signal`. This procedure is described in “Creating Packages that Support CSC Definitions” on page 7-8.

Add your own object properties and class initialization if desired. For each of your classes, select the **Create your own custom storage classes for this class** option.

Create a Custom Attributes Class for Your CSC (Optional)

If you have instance-specific properties that are relevant only to your CSC, you should use the Simulink Data Class Designer to create a *custom attributes class* for the package. A custom attributes class is a subclass of `Simulink.CustomStorageClassAttributes`. The name, type, and default

value properties you set for the custom attributes class define the user view of instance-specific properties.

For example, the `ExportToFile` custom storage class requires that you set the `RTWInfo.CustomAttributes.HeaderFile` property to specify a `.h` file used for exporting each piece of data. See “Simulink Package Custom Storage Classes” on page 7-6 for further information on instance-specific properties.

Write TLC Code for Your CSC

The next step is to write TLC code that implements code generation for data of your new custom storage class. A template TLC file is provided for this purpose. To create your TLC code, follow these steps:

- 1 Create a `tlc` directory inside your package’s `@directory` (if it does not already exist). The naming convention to follow is

```
@PackageName/tlc
```

- 2 Copy `TEMPLATE_v1.tlc` (or another CSC template) from `matlabroot/toolbox/rtw/targets/ecoder/csc_templates` into your `tlc` directory to use as a starting point for defining your custom storage class.
- 3 Write your TLC code, following the comments in the CSC template file. Comments describe how to specify code generation for data of your custom storage class (for example, how data structures are to be declared, defined, and whether they are accessed by value or by reference).

Alternatively, you can copy a custom storage class TLC file from another existing package as a starting point for defining your custom storage class.

Register Custom Storage Class Definitions

After you have created a package for your new custom storage class and written its associated TLC code, you must register your class definitions with the Custom Storage Class Designer, using its advanced mode.

The advanced mode supports selection of an additional storage class **Type**, designated **Other**. The **Other** type is designed to support special CSC types that cannot be accommodated by the standard **Unstructured** and

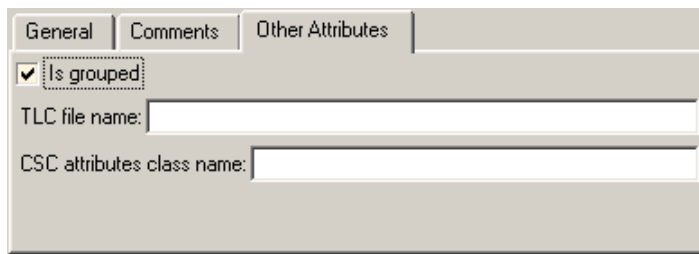
FlatStructure custom storage class types. The **Other** type cannot be assigned to a CSC except when the Custom Storage Class Designer is in advanced mode.

To register your class definitions:

- 1 Launch the Custom Storage Class Designer in advanced mode by typing the following command at the MATLAB prompt:

```
cscdesigner -advanced
```

- 2 Select your package and create a new custom storage class.
- 3 Set the **Type** of the custom storage class to **Other**. Note that when you do this, the **Other Attributes** pane is displayed. This pane is visible only for CSCs whose **Type** is set to **Other**.



If you specify a customized package, additional options, as defined by the package, also appear on the **Other Attributes** pane.

- 4 Set the properties shown on the **Other Attributes** pane. The properties are:
 - **Is grouped:** Select this option if you intend to combine multiple data objects of this CSC into a single variable in the generated code. (for example, a struct).
 - **TLC file name:** Enter the name of the TLC file corresponding to this custom storage class. The location of the file is assumed to be in the `/t1c` subdirectory for the package, so you should not enter the path to the file.
 - **CSC attributes class name:** (optional) If you created a custom attributes class corresponding to this custom storage class, enter the full

name of the custom attributes class. (see “Create a Custom Attributes Class for Your CSC (Optional)” on page 7-62).

- 5** Set the remaining properties on the **General** and **Comments** panes based on the layout of the data that you wish to generate (as defined in your TLC file).

GetSet Custom Storage Class for Data Store Memory

In this section...
“Overview” on page 7-66
“GetSet CSC Properties” on page 7-66
“Using the GetSet CSC” on page 7-67
“GetSet CSC Restrictions” on page 7-67
“GetSet Custom Storage Class Example” on page 7-68

Overview

GetSet is a built-in advanced custom storage class that generates specialized function calls to read from (get) and write to (set) the memory associated with a Data Store Memory block that is read and written many times in a single model. See “Working with Data Stores” for information about data stores and the Data Store Memory block, and for information about advanced CSCs.

The GetSet custom storage class is designed primarily for use with the state of the Data Store Memory block. However, GetSet is capable of handling signals other than data stores, and is supported for the outputs of most built-in blocks provided by The MathWorks. For more about the definition of the GetSet storage class, look at its associated TLC code in the file:

```
matlabroot\toolbox\simulink\simulink\@Simulink\tlc\GetSet.tlc
```

GetSet CSC Properties

The next table summarizes the instance-specific properties of the GetSet storage class:

Property	Description
GetFunction	String that specifies the name of a function call to read data.
SetFunction	String that specifies the name of a function call to write data.
HeaderFile (optional)	String that specifies the name of a header (.h) file to add as an <code>#include</code> in the generated code.

For example, if the `GetFunction` of signal `X` is specified as `'get_X'` then the generated code calls `get_X()` wherever the value of `X` is used. Similarly, if the `SetFunction` of signal `X` is specified as `'set_X'` then the generated code calls `set_X(value)` wherever the value of `X` is assigned.

Using the GetSet CSC

The `GetSet` storage class cannot be represented by the standard `Unstructured` or `FlatStructure` custom storage class types, so it is an advanced CSC, as described in . To access the CSC definition for `GetSet`, you must launch Custom Storage Class designer in advanced mode:

```
cscdesigner -advanced
```

If you omit the `HeaderFile` property for a `GetSet` data object, you must specify a header file by an alternative means, such as the **Header file** field of the **Real-Time Workshop/Custom Code** pane of the Configuration Parameters dialog box. Otherwise, the generated code might not compile or might function improperly.

For wide signals, an additional index argument is passed, so the calls to the `get` and `set` functions are `get_X(idx)` and `set_X(idx, value)` respectively.

GetSet CSC Restrictions

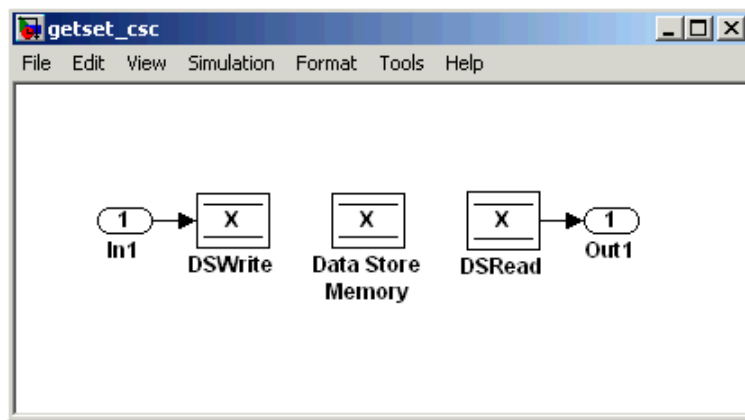
- The `GetSet` supports only signals of noncomplex data types.
- Some built-in blocks do not directly support `GetSet`.
- User-written S-functions do not directly support `GetSet`.

To use GetSet with a nonsupporting built-in block or a user-written S-function:

- 1 Insert a Signal Conversion block at the output of the block or function.
- 2 Select the Signal Conversion Block's **Exclude this block from 'Block reduction' optimization** property.
- 3 Assign the GetSet storage class to the output of the Signal Conversion block.

GetSet Custom Storage Class Example

The model below contains a Data Store Memory block that resolves to the Simulink signal object X:



The following specifications configure the signal object X to use the GetSet custom storage class:

```
X = Simulink.Signal;
X.RTWInfo.StorageClass           = `Custom';
X.RTWInfo.CustomStorageClass    = `GetSet';
X.RTWInfo.CustomAttributes.GetFunction = `get_X';
X.RTWInfo.CustomAttributes.SetFunction = `set_X';
X.RTWInfo.CustomAttributes.HeaderFile = `user_file.h';
```

The GetSet CSC appears as follows in the code generated for the model:

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

void getset_csc_step(void)
{
    /* local block i/o variables */
    real_T rtb_DSRead_o;

    /* DataStoreWrite: '<Root>/DSWrite' incorporates:
     *   Inport: '<Root>/In1'
     */
    set_X(getset_csc_U.In1);

    /* DataStoreRead: '<Root>/DSRead' */
    rtb_DSRead_o = get_X();

    /* Outport: '<Root>/Out1' */
    getset_csc_Y.Out1 = rtb_DSRead_o;
}
```

Note that the code uses a local variable `rtb_DSRead_o` rather than multiple calls to the `get_X` function. This technique increases code efficiency and ensures that the value does not change within a simulation step.

Custom Storage Class Implementation

You can skip this section unless you want to ship custom storage class definitions in an uneditable format, or you intend to bypass the Custom Storage Class designer and work directly with files that contain custom storage class definitions.

The file that defines a package's custom storage classes is called a *CSC registration file*. The file is always named `csc_registration` and resides in the *@package* directory that defines the package. A CSC registration file can be a P-file (`csc_registration.p`) or a MATLAB file (`csc_registration.m`). A built-in package defines custom storage classes in both a P-file and a functionally equivalent MATLAB file. A user-defined package initially defines custom storage classes only in a MATLAB file.

P-files take precedence over MATLAB files, so when MATLAB looks for a package's CSC registration file and finds both a P-file and a MATLAB file, MATLAB loads the P-file and ignores the MATLAB file. All capabilities and tools, including the Custom Storage Class Designer, then use the CSC definitions stored in the P-file. P-files cannot be edited, so all CSC Designer editing capabilities are disabled for CSCs stored in a P-file. If no P-file exists, MATLAB loads CSC definitions from the MATLAB file. MATLAB files are editable, so all CSC Designer editing capabilities are enabled for CSCs stored in a MATLAB file.

Because CSC definitions for a built-in package exist in both a P-file and a MATLAB file, they are uneditable. You can make the definitions editable by deleting the P-file, but The MathWorks strongly discourages modifying built-in CSC registration files or any other files under `matlabroot`. The preferred technique is to create user-defined packages, data classes, and custom storage classes, as described in “Subclassing Simulink Data Classes” and this chapter.

The CSC Designer saves CSC definitions for user-defined packages in a MATLAB file, so the definitions are editable. You can make the definitions uneditable by using the `pcode` function to create an equivalent P-file, which will then shadow the MATLAB file. However, you should preserve the MATLAB file if you may need to make further changes, because you cannot modify CSC definitions that exist only in a P-file.

You can also use tools or techniques other than the Custom Storage Class Designer to create and edit MATLAB files that define CSCs. However, The MathWorks discourages this practice, which is vulnerable to syntax errors and can give unexpected results. When MATLAB finds an older P-file that shadows a newer MATLAB file, it displays a warning in the MATLAB Command Window.

Custom Storage Class Limitations

- The Fcn block does not support parameters with custom storage class in code generation.
- For CSCs in models that use referenced models (see “Referencing a Model”):
 - If data is assigned a grouped CSC, such as `Struct` or `Bitfield`, the CSC’s **Data scope** property must be `Imported` and the data declaration must be provided in a user-supplied header file. See “Grouped Custom Storage Classes” on page 7-61 for more information about grouped CSCs.
 - If data is assigned an ungrouped CSC, such as `Const`, and the data’s **Data scope** property is `Exported`, its **Header file** property must be unspecified. This results in the data being exported with the standard header file, `model.h`. Note that for ungrouped data, the **Data scope** and **Header file** properties are either specified by the selected CSC, or as one of the data object’s instance-specific properties.

Custom Storage Classes Prior to R2009a

If you save a model that uses embedded signal objects in a release prior to R2009a, the saved model omits the embedded signal objects. The saved model contains the same information that it would have contained if the embedded signal objects had never existed, and output ports have no `SignalObject` property. See “Applying a CSC Using an Embedded Signal Object” on page 7-43 for information about embedded signal objects.

Custom Storage Classes Prior to Release 14

In this section...
“Introduction” on page 7-74
“Simulink.CustomParameter Class” on page 7-74
“Simulink.CustomSignal Class” on page 7-75
“Instance-Specific Attributes for Older Storage Classes” on page 7-78
“Assigning a Custom Storage Class to Data” on page 7-79
“Code Generation with Older Custom Storage Classes” on page 7-80
“Compatibility Issues for Older Custom Storage Classes” on page 7-81

Introduction

In releases prior to Real-Time Workshop Embedded Coder version 4.0 (MATLAB Release 14), custom storage classes were implemented with special `Simulink.CustomSignal` and `Simulink.CustomParameter` classes. This section describes these older classes.

Note Models that use the `Simulink.CustomSignal` and `Simulink.CustomParameter` classes continue to operate correctly. The current CSCs support a superset of the functions of the older classes. Therefore, you should consider using the `Simulink.Signal` and `Simulink.Parameter` classes instead (see “Compatibility Issues for Older Custom Storage Classes” on page 7-81).

Simulink.CustomParameter Class

This class is a subclass of `Simulink.Parameter`. Objects of this class have expanded `RTWInfo` properties. The properties of `Simulink.CustomParameter` objects are:

- `RTWInfo.StorageClass`. This property should always be set to the default value, `Custom`.

- `RTWInfo.CustomStorageClass`. This property takes on one of the enumerated values described in the tables below. This property controls the generated storage declaration and code for the object.
- `RTWInfo.CustomAttributes`. This property defines additional attributes that are exclusive to the class, as described in “Instance-Specific Attributes for Older Storage Classes” on page 7-78.
- `Value`. This property is the numeric value of the object, used as an initial (or inlined) parameter value in generated code.

Simulink.CustomSignal Class

This class is a subclass of `Simulink.Signal`. Objects of this class have expanded `RTWInfo` properties. The properties of `Simulink.CustomSignal` objects are:

- `RTWInfo.StorageClass`. This property should always be set to the default value, `Custom`.
- `RTWInfo.CustomStorageClass`. This property takes on one of the enumerated values described in the tables below. This property controls the generated storage declaration and code for the object.
- `RTWInfo.CustomAttributes`. This optional property defines additional attributes that are exclusive to the storage class, as described in “Instance-Specific Attributes for Older Storage Classes” on page 7-78.

The following tables summarize the predefined custom storage classes for `Simulink.CustomSignal` and `Simulink.CustomParameter` objects. The entry for each class indicates

- Name and purpose of the class.
- Whether the class is valid for parameter or signal objects. For example, you can assign the storage class `Const` to a parameter object. This storage class is not valid for signals, however, since signal data (except for the case of invariant signals) is not constant.
- Whether the class is valid for complex data or nonscalar (wide) data.
- Data types supported by the class.

The first three classes, shown in `Const`, `ConstVolatile`, and `Volatile` Storage Classes (Prior to Release 14) on page 7-76, insert type qualifiers in the data declaration.

Const, ConstVolatile, and Volatile Storage Classes (Prior to Release 14)

Class Name	Purpose	Parameters	Signals	Data Types	Complex	Wide
Const	Use <code>const</code> type qualifier in declaration	Y	N	any	Y	Y
ConstVolatile	Use <code>const volatile</code> type qualifier in declaration	Y	N	any	Y	Y
Volatile	Use <code>volatile</code> type qualifier in declaration	Y	Y	any	Y	Y

The second set of three classes, shown in `ExportToFile`, `ImportFromFile`, and `Internal` Storage Classes (Prior to Release 14) on page 7-76, handles issues of data scope and file partitioning.

ExportToFile, ImportFromFile, and Internal Storage Classes (Prior to Release 14)

Class Name	Purpose	Parameters	Signals	Data Types	Complex	Wide
ExportToFile	Generate and include files, with user-specified name, containing global variable declarations and definitions	Y	Y	any	Y	Y

ExportToFile, ImportFromFile, and Internal Storage Classes (Prior to Release 14) (Continued)

Class Name	Purpose	Parameters	Signals	Data Types	Complex	Wide
ImportFromFile	Include predefined header files containing global variable declarations	Y	Y	any	Y	Y
Internal	Declare and define global variables whose scope is limited to Real-Time Workshop generated code	Y	Y	any	Y	Y

The final three classes, shown in BitField, Define, and Struct Storage Classes (Prior to Release 14) on page 7-77, specify the data structure or construct used to represent the data.

BitField, Define, and Struct Storage Classes (Prior to Release 14)

Class Name	Purpose	Parameters	Signals	Data types	Complex	Wide
BitField	Embed Boolean data in a named bit field	Y	Y	Boolean	N	N
Define	Represent parameters with a #define macro	Y	N	any	N	N

BitField, Define, and Struct Storage Classes (Prior to Release 14) (Continued)

Class Name	Purpose	Parameters	Signals	Data types	Complex	Wide
Struct	Embed data in a named struct to encapsulate sets of data	Y	Y	any	N	Y

Instance-Specific Attributes for Older Storage Classes

Some custom storage classes have attributes that are exclusive to the class. These attributes are made visible as members of the `RTWInfo.CustomAttributes` field. For example, the `BitField` class has a `BitFieldName` attribute (`RTWInfo.CustomAttributes.BitFieldName`).

Additional Properties of Custom Storage Classes (Prior to Release 14) on page 7-78 summarizes the storage classes with additional attributes, and the meaning of those attributes. Attributes marked optional have default values and may be left unassigned.

Additional Properties of Custom Storage Classes (Prior to Release 14)

Storage Class Name	Additional Properties	Description	Optional (has default)
ExportToFile	FileName	String. Defines the name of the generated header file within which the global variable declaration should reside. If unspecified, the declaration is placed in <code>model_export.h</code> by default.	Y
ImportFromFile	FileName	String. Defines the name of the generated header file which to be used in <code>#include</code> directive.	N

Additional Properties of Custom Storage Classes (Prior to Release 14) (Continued)

Storage Class Name	Additional Properties	Description	Optional (has default)
ImportFromFile	IncludeDelimiter	Enumerated. Defines delimiter used for filename in the #include directive. Delimiter is either double quotes (for example, #include "vars.h") or angle brackets (for example, #include <vars.h>). The default is quotes.	Y
BitField	BitFieldName	String. Defines name of bit field in which data is embedded; if unassigned, the name defaults to rt_BitField.	Y
Struct	StructName	String. Defines name of the struct in which data is embedded; if unassigned, the name defaults to rt_Struct.	Y

Assigning a Custom Storage Class to Data

You can create custom parameter or signal objects from the MATLAB command line. For example, the following commands create a custom parameter object `p` and a custom signal object `s`:

```
p = Simulink.CustomParameter
s = Simulink.CustomSignal
```

After creating the object, set the `RTWInfo.CustomStorageClass` and `RTWInfo.CustomAttributes` fields. For example, the following commands sets these fields for the custom parameter object `p`:

```
p.RTWInfo.CustomStorageClass = 'ExportToFile'
p.RTWInfo.CustomAttributes.FileName = 'testfile.h'
```

Finally, make sure that the `RTWInfo.StorageClass` property is set to `Custom`. If you inadvertently set this property to some other value, the custom storage properties are ignored.

Code Generation with Older Custom Storage Classes

The procedure for generating code with data objects that have a custom storage class is similar to the procedure for code generation using Simulink data objects that have built-in storage classes. If you are unfamiliar with this procedure, see the discussion of Simulink data objects in the “Defining Data Representation and Storage for Code Generation” section of the Real-Time Workshop documentation.

To generate code with custom storage classes, you must

- 1 Create one or more data objects of class `Simulink.CustomParameter` or `Simulink.CustomSignal`.
- 2 Set the custom storage class property of the objects, as well as the class-specific attributes (if any) of the objects.
- 3 Reference these objects as block parameters, signals, block states, or Data Store memory.

When generating code from a model employing custom storage classes, make sure that the **Ignore custom storage classes** option is *not* selected. This is the default for the Real-Time Workshop Embedded Coder software.

When **Ignore custom storage classes** is selected:

- Objects with custom storage classes are treated as if their storage class attribute is set to `Auto`.
- The storage class of signals that have custom storage classes is not displayed on the signal line, even if the **Storage class** option of the Simulink model editor **Format** menu is selected.

Ignore custom storage classes lets you switch to a target that does not support CSCs, such as the generic real-time target (GRT), without having to reconfigure your parameter and signal objects.

When using the Real-Time Workshop Embedded Coder software, you can control the **Ignore custom storage classes** option with the check box in the **Real-Time Workshop** pane of the Configuration Parameters dialog box.

If you are using a target that does not have a check box for this option (such as a custom target) you can enter the option directly into the **TLC options** field in the **Real-Time Workshop** pane of the Configuration Parameters dialog box. The following example turns the option on:

```
-aIgnoreCustomStorageClasses=1
```

Compatibility Issues for Older Custom Storage Classes

In Release 14, the full functionality of the `Simulink.CustomSignal` and `Simulink.CustomParameter` classes was added to the `Simulink.Signal` and `Simulink.Parameter` classes. You should consider replacing the use of `Simulink.CustomSignal` and `Simulink.CustomParameter` objects by using equivalent `Simulink.Signal` and `Simulink.Parameter` objects.

If you prefer, you can continue to use the `Simulink.CustomSignal` and `Simulink.CustomParameter` classes in the current release. Note that the following changes have been implemented in these classes:

- The `Internal` storage class has been removed from the enumerated values of the `RTWInfo.CustomStorageClass` property. `Internal` storage class is no longer supported.
- For the `ExportToFile` and `ImportFromFile` storage classes, the `RTWInfo.CustomAttributes.FileName` and `RTWInfo.CustomAttributes.IncludeDelimiter` properties have been obsoleted and combined into a single property, `RTWInfo.CustomAttributes.HeaderFile`. When specifying a header file, include both the filename and the required delimiter as you want them to appear in generated code, as in the following example:

```
myobj.RTWInfo.CustomAttributes.HeaderFile = '<myheader.h>';
```

- Prior to Release 14, user-defined CSCs were created by designing custom packages that included the CSC definitions. This technique for creating

CSCs is obsolete; see “Creating Packages that Support CSC Definitions” on page 7-8 for a description of the current procedure, which is much simpler.

If you designed your own custom packages containing CSCs prior to Release 14 you should convert them to Release 14 CSCs. The conversion procedure is described in the next section, “Converting Older Packages to Use CSC Registration Files” on page 7-82.

Converting Older Packages to Use CSC Registration Files

A Simulink data class package can be associated with one or more CSC definitions. In Release 14, the linkage between a set of CSC definitions and a package is formed when a CSC registration file (`csc_registration.m`) is located in the package directory.

Prior to Release 14, user-defined CSCs were created by designing custom packages that included the CSC definitions as part of the package.

The Simulink Data Class Designer supports conversion of older packages to the use of CSC registration files. When such a package is selected in Data Class Designer, a special conversion button is displayed on the **Custom Storage Classes** pane. This button lets you invoke a conversion procedure; you can then write out all files and directories required to define the package, including a CSC registration file. To convert a package:

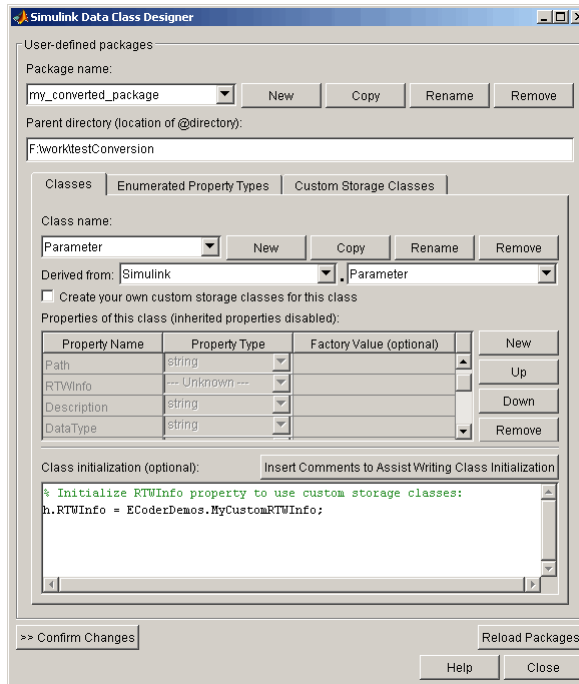
- 1 You should make a complete backup copy of the package directory before converting the package. After backing up the directory, remove the `@` prefix from the backup directory name and make sure that the backup directory is not on the MATLAB path.
- 2 Open the Simulink Data Class Designer by choosing **Tools > Data Class Designer** in the model window, or typing the following at the MATLAB command prompt:

```
sldataclassdesigner
```

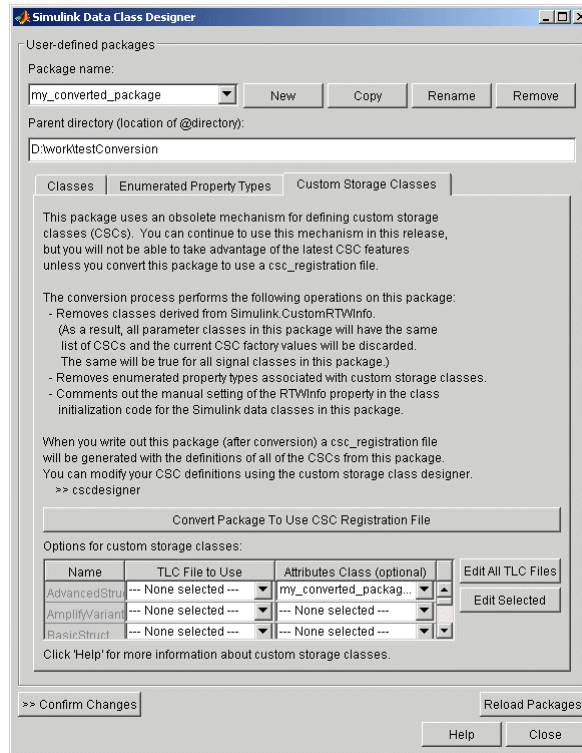
- 3 The Data Class Designer loads all packages that exist on the MATLAB path. Select the package to be converted from the **Package name** menu. Then, click **OK**.

- 4 If you want to store the converted package in a different directory than the original package, enter the desired path in the **Parent directory** field. This step is optional.

The figure below shows the package `my_converted_package`. The package definition is stored in `d:\work\testConversion`.



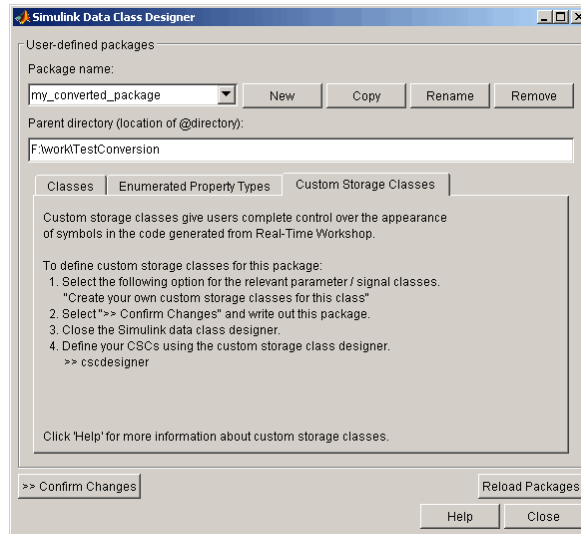
- 5 Click on the **Custom Storage Classes** pane. The pane displays a message indicating that the package contains obsolete CSC definitions, as shown in this figure.



Below the message text, the pane also contains a button captioned **Convert Package to Use CSC Registration File**. This button invokes a script that converts the package to use a CSC registration file.

Note that this button does not actually create the CSC registration file. That happens when the package files are written out, as described below.

- 6 Click **Convert Package to Use CSC Registration File**. After conversion, the appearance of the pane changes, as shown below.



7 Click **Confirm Changes**. In the **Confirm Changes** pane, select the package you converted. Add the parent directory to the MATLAB path if necessary. Then, click **Write Selected**.

8 Click **Close**.

9 You can now view and edit the CSCs belonging to your package in the Custom Storage Class Designer. To do so, type the following command at the MATLAB prompt:

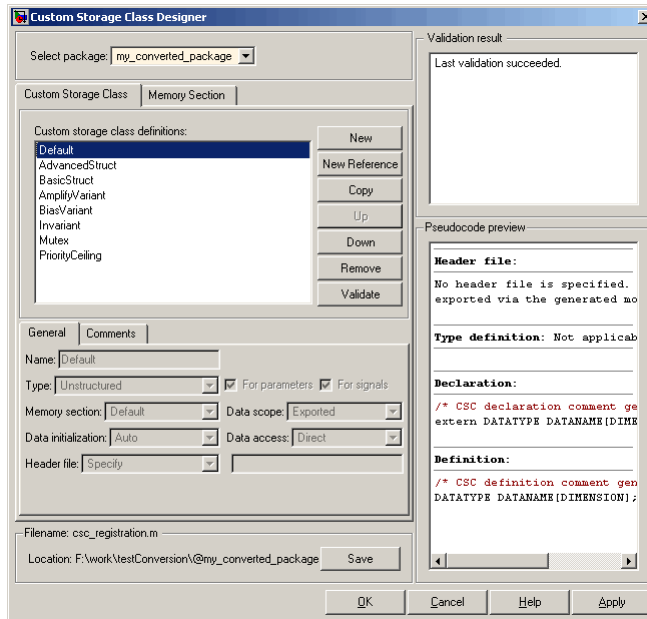
```
cscdesigner
```

Note You must launch the CSC Designer with the `-advanced` motion to edit the attributes of old CSCs because they are defined with user-defined TLC files.

The Custom Storage Class Designer loads all packages that have a CSC registration file.

10 Select your converted package from the **Select package** menu.

The figure below shows the Custom Storage Class Designer displaying the CSCs defined in the package `my_converted_package`. See “Designing Custom Storage Classes and Memory Sections” on page 7-12 for a description of the operation of the Custom Storage Class Designer.



Note All user-defined CSCs created prior to Release 14 are defined with their own TLC code. As a result, after conversion, the **Type** is set to **Other** (as opposed to **Unstructured** or **FlatStructure**). See “Defining Advanced Custom Storage Class Types” on page 7-62 for more information.

- 11 Restart your MATLAB session to ensure that your converted package is accessible.

Inserting Comments and Pragmas in Generated Code

- “Introduction to Memory Sections” on page 8-2
- “Requirements for Defining Memory Sections” on page 8-4
- “Defining Memory Sections” on page 8-7
- “Configuring Memory Sections” on page 8-11
- “Applying Memory Sections” on page 8-12
- “Examples of Generated Code with Memory Sections” on page 8-20
- “Model-Level Data Structures” on page 8-22
- “Memory Section Limitation” on page 8-25

Introduction to Memory Sections

In this section...
“Overview” on page 8-2
“Memory Sections Demo” on page 8-2
“Additional Information” on page 8-2

Overview

The Real-Time Workshop Embedded Coder software provides a memory section capability that allows you to insert comments and pragmas into the generated code for

- Data in custom storage classes
- Model-level functions
- Model-level internal data
- Subsystem functions
- Subsystem internal data

Pragmas inserted into generated code can surround

- A contiguous block of function or data definitions
- Each function or data definition separately

When pragmas surround each function or data definition separately, the text of each pragma can contain the name of the definition to which it applies.

Memory Sections Demo

To see a demo of memory sections, type `rtwdemo_memsec` in the MATLAB Command Window.

Additional Information

See the following for additional information relating to memory sections:

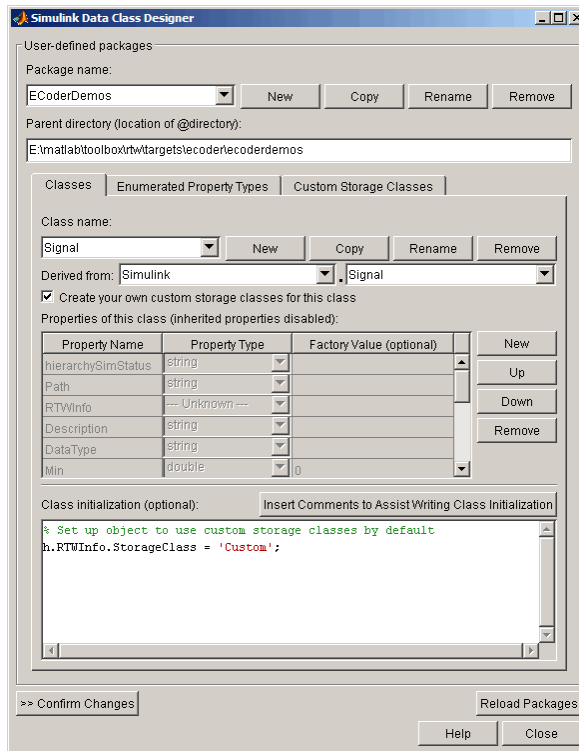
- Simulink data types, packages, data classes, and data objects:
 - “Working with Data” in the Simulink documentation
- Real-Time Workshop data structures and storage classes:
 - “Defining Data Representation and Storage for Code Generation” in the Real-Time Workshop documentation
- Real-Time Workshop Embedded Coder custom storage classes:
 - Chapter 7, “Creating and Using Custom Storage Classes” in the Real-Time Workshop Embedded Coder documentation
- Fine-tuned optimization of generated code for functions or data:
 - The *Real-Time Workshop Target Language Compiler* documentation

Requirements for Defining Memory Sections

Before you can define memory sections, you must do the following:

- 1 Set the Simulink model's code generation target to an embedded target such as `ert.tlc`.
- 2 If you need to create packages, specify package properties, or create classes, including custom storage classes, choose **Tools > Data Class Designer** in the model window.

A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the Simulink Data Class Designer appears. The Data Class Designer initially looks like this:



Complete instructions for using the Data Class Designer appear in “Subclassing Simulink Data Classes” in the Simulink documentation. See also the instructions that appear when you click the **Custom Storage Classes** tab.

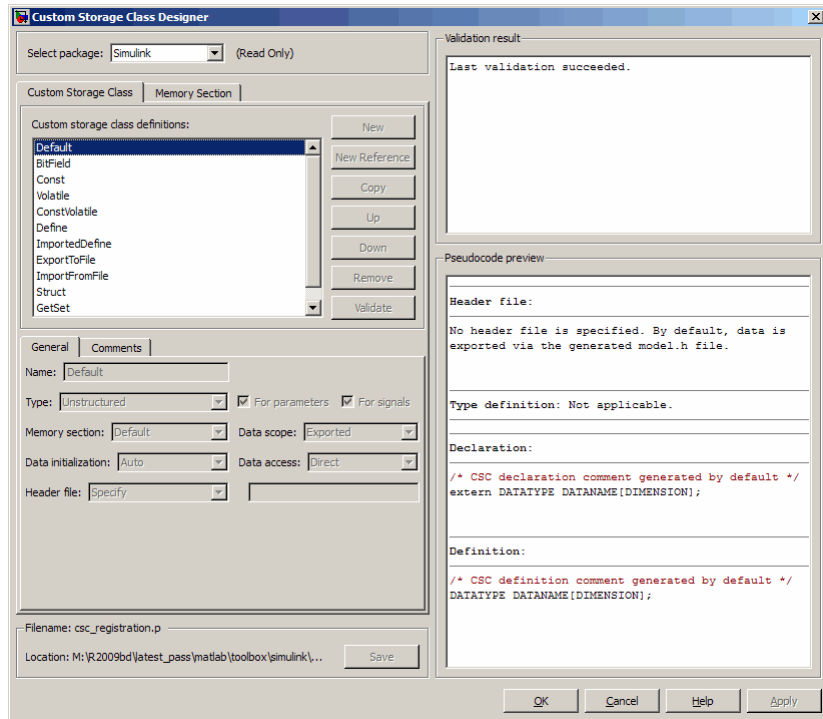
- 3** If you need to specify custom storage class properties,
 - a** Choose **View > Model Explorer** in the model window.

The Model Explorer appears.

- b** Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

- c** Select the **Custom Storage Class** tab. The **Custom Storage Class** pane initially looks like this:



- d** Use the **Custom Storage Class** pane as needed to select a writable package and specify custom storage class properties. Instructions for using this pane appear in “Designing Custom Storage Classes and Memory Sections” on page 7-12.

Defining Memory Sections

In this section...

“Editing Memory Section Properties” on page 8-7

“Specifying the Memory Section Name” on page 8-9

“Specifying a Qualifier for Custom Storage Class Data Definitions” on page 8-9

“Specifying Comment and Pragma Text” on page 8-9

“Surrounding Individual Definitions with Pragmas” on page 8-9

“Including Identifier Names in Pragmas” on page 8-10

Editing Memory Section Properties

After you have satisfied the requirements in “Requirements for Defining Memory Sections” on page 8-4, you can define memory sections and specify their properties. To create new memory sections or specify memory section properties,

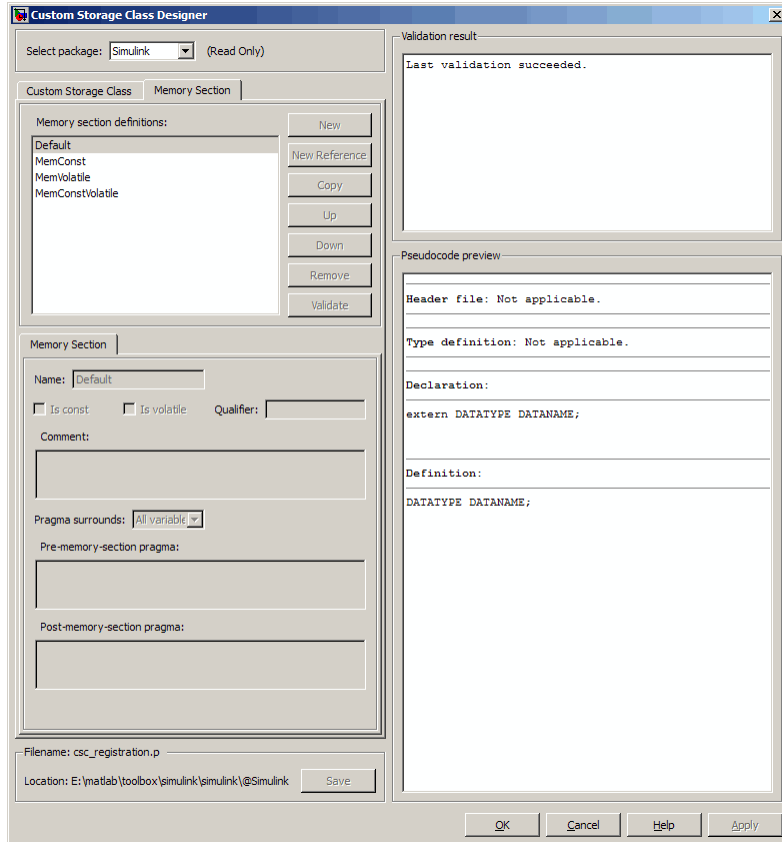
- 1 Choose **View > Model Explorer** in the model window.

The Model Explorer appears.

- 2 Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

- 3 Click the **Memory Section** tab of the Custom Storage Class Designer. The **Memory Section** pane initially looks like this:



- 4** If you intend to create or change memory section definitions, use the **Select package** field to select a writable package.

The rest of this section assumes that you have selected a writable package, and describes the use of the **Memory section** subpane on the lower left. For descriptions of the other subpanes, instructions for validating memory section definitions, and other information, see “Creating and Editing Memory Section Definitions” on page 7-31.

Specifying the Memory Section Name

To specify the name of a memory section, use the **Name** field. A memory section name must be a legal MATLAB identifier.

Specifying a Qualifier for Custom Storage Class Data Definitions

To specify a qualifier for custom storage class data definitions in a memory section, enter the components of the qualifier below the **Name** field.

- To specify `const`, check **Is const**.
- To specify `volatile`, check **Is volatile**.
- To specify anything else (e.g., `static`), enter the text in the **Qualifier** field.

The qualifier will appear in generated code with its components in the same left-to-right order in which their definitions appear in the dialog box. A preview appears in the **Pseudocode preview** subpane on the lower right.

Note Specifying a qualifier affects only custom storage class data definitions. The code generator omits the qualifier from any other category of definition.

Specifying Comment and Pragma Text

To specify a comment, prepragma, or postpragma for a memory section, enter the text in the appropriate edit boxes on the left side of the Custom Storage Class Designer. These boxes accept multiple lines separated by ordinary Returns.

Surrounding Individual Definitions with Pragmas

If the **Pragma surrounds** field for a memory section specifies `Each variable`, the code generator will surround each definition in a contiguous block of definitions with the comment, prepragma, and postpragma defined for the section. This behavior occurs with all categories of definitions.

If the **Pragma surrounds** field for a memory section specifies `All variables`, the code generator will insert the comment and prepragma for the

section before the first definition in a contiguous block of custom storage class data definitions, and the postpragma after the last definition in the block.

Note Specifying `All variables` affects only custom storage class data definitions. For any other category of definition, the code generator surrounds each definition separately regardless of the value of **Pragma surrounds**.

Including Identifier Names in Pragmas

When pragmas surround each separate definition in a contiguous block, you can include the string `%<identifier>` in a pragma. The string must appear without surrounding quotes.

- When `%<identifier>` appears in a prepragma, the code generator will substitute the identifier from the subsequent function or data definition.
- When `%<identifier>` appears in a postpragma, the code generator will substitute the identifier from the previous function or data definition.

You can use `%<identifier>` with pragmas *only* when pragmas to surround each variable. The `Validate` phase will report an error if you violate this rule.

Note Although `%<identifier>` looks like a TLC variable, it is not: it is just a keyword that directs the code generator to substitute the applicable data definition identifier when it outputs a pragma. TLC variables cannot appear in pragma specifications in the **Memory Section** pane.

Configuring Memory Sections

You can configure a model such that the generated code includes comments and pragmas for

- Data defined in custom storage classes
- Internal data not defined in custom storage classes
- Model-level functions
- Atomic subsystem functions with or without separate data

To...	Select...
Specify the package that contains memory sections that you want to apply	The name of a package for Package . Click Refresh package list to refresh the list of available packages in your configuration.
Apply memory sections to initialize/start and terminate functions	A value for Initialize/Terminate .
Apply memory sections to step, run-time initialization, derivative, enable, and disable functions	A value for Execution .
Apply memory sections to constant parameters, constant, block I/O, or zero representation	A value for Constants .
Apply memory sections to root inputs or outputs	A value for Inputs/Outputs .
Apply memory sections to block I/O, Dwork vectors, run-time models, zero-crossings	A value for Internal data .
Apply memory sections to parameters	A value for Parameters .

The interface checks whether the specified package is on the MATLAB path and that the selected memory sections are in the package. The results of this validation appear in the field **Validation results**.

Applying Memory Sections

In this section...
“Assigning Memory Sections to Custom Storage Classes” on page 8-12
“Applying Memory Sections to Model-Level Functions and Internal Data” on page 8-14
“Applying Memory Sections to Atomic Subsystems” on page 8-16

Assigning Memory Sections to Custom Storage Classes

To assign a memory section to a custom storage class,

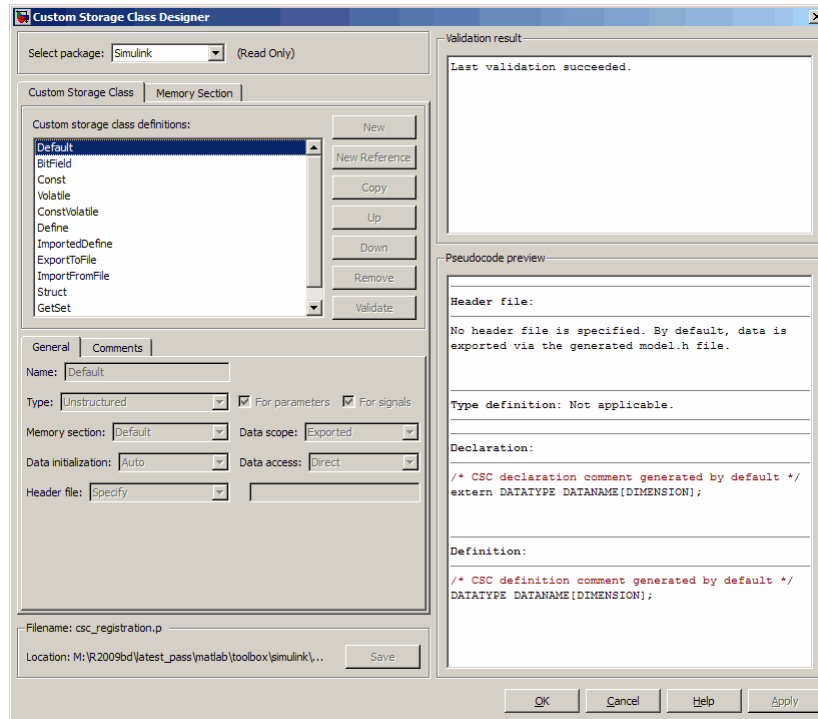
- 1 Choose **View > Model Explorer** in the model window.

The Model Explorer appears.

- 2 Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

- 3 Select the **Custom Storage Class** tab. The **Custom Storage Class** pane initially looks like this:



- 4** Use the **Select package** field to select a writable package. The rest of this section assumes that you have selected a writable package.
- 5** Select the desired custom storage class in the **Custom storage class definitions** pane.
- 6** Select the desired memory section from the **Memory section** pull-down.
- 7** Click **Apply** to apply changes to the open copy of the model; **Save** to apply changes and save them to disk; or **OK** to apply changes, save changes, and close the Custom Storage Class Designer.

Generated code for all data definitions in the specified custom storage class will be enclosed in the pragmas of the specified memory section. The pragmas can surround contiguous blocks of definitions or each definition separately, as described in “Surrounding Individual Definitions with Pragmas” on page 8-9.

For more information, see “Creating Packages that Support CSC Definitions” on page 7-8.

Applying Memory Sections to Model-Level Functions and Internal Data

When using the Real-Time Workshop Embedded Coder software, you can apply memory sections to the following categories of model-level functions:

Function Category	Function Subcategory
Initialize/Terminate functions	Initialize/Start
	Terminate
Execution functions	Step functions
	Run-time initialization
	Derivative
	Enable
	Disable

When using the Real-Time Workshop Embedded Coder software, you can apply memory sections to the following categories of internal data:

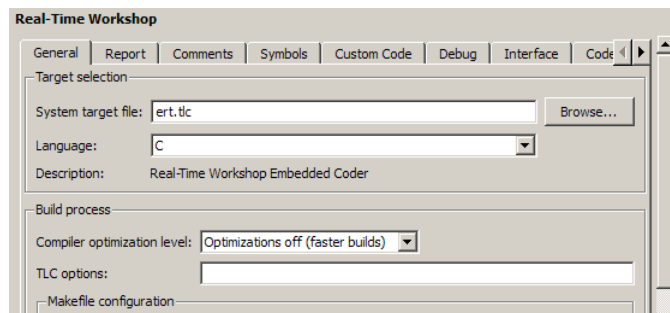
Data Category	Data Definition	Data Purpose
Constants	<i>model_cP</i>	Constant parameters
	<i>model_cB</i>	Constant block I/O
	<i>model_Z</i>	Zero representation
Input/Output	<i>model_U</i>	Root inputs
	<i>model_Y</i>	Root outputs

Data Category	Data Definition	Data Purpose
Internal data	<i>model_B</i>	Block I/O
	<i>model_D</i>	D-work vectors
	<i>model_M</i>	Run-time model
	<i>model_Zero</i>	Zero-crossings
Parameters	<i>model_P</i>	Parameters

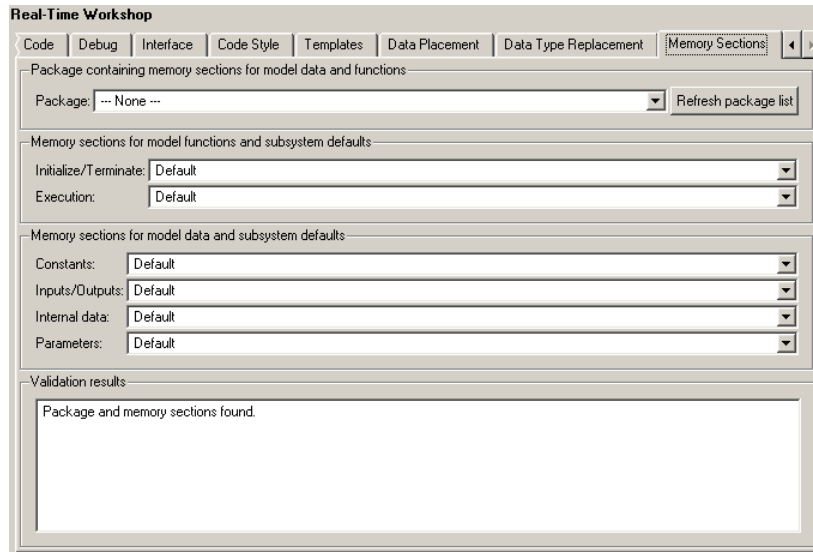
Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems *except* atomic subsystems that contain overriding memory section specifications, as described in “Applying Memory Sections to Atomic Subsystems” on page 8-16.

To specify memory sections for model-level functions or internal data,

- 1 Open the Model Explorer and select **Configuration (Active)** > **Real-Time Workshop** > **General**. (Alternatively, choose **Simulation** > **Configuration Parameters** in the model window.)
- 2 Ensure that the **System target file** is an ERT target, such as `ert.tlc`.



- 3 Select the **Memory Sections** tab. The **Memory Sections** pane looks like this:



- 4 Initially, the **Package** field specifies ---None--- and the pull-down lists only built-in packages. If you have defined any packages of your own, click **Refresh package list**. This action adds all user-defined packages on your search path to the package list.
- 5 In the **Package** pull-down, select the package that contains the memory sections that you want to apply.
- 6 In the pull-down for each category of internal data and model-level function, specify the memory section (if any) that you want to apply to that category. Accepting or specifying **Default** omits specifying any memory section for that category.
- 7 Click **Apply** to save any changes to the package and memory section selections.

Applying Memory Sections to Atomic Subsystems

For any atomic subsystem whose generated code format is **Function** or **Reusable Function**, you can specify memory sections for functions and

internal data that exist in that code format. Such specifications override any model-level memory section specifications. Such overrides apply only to the atomic subsystem itself, not to any subsystems within it. Subsystems of an atomic subsystem inherit memory section specifications from the containing model, *not* from the containing atomic subsystem.

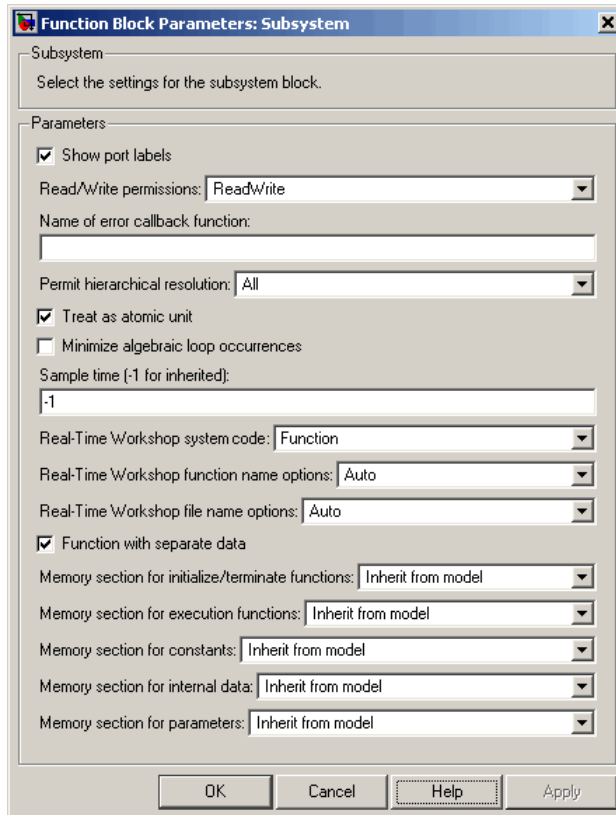
To specify memory sections for an atomic subsystem,

- 1** Right-click the subsystem in the model window.
- 2** Choose **Subsystem Parameters** from the context menu. The Function Block Parameters: *Subsystem* dialog box appears.
- 3** Ensure that **Treat as atomic unit** is checked. Otherwise, you cannot specify memory sections for the subsystem.

For an atomic system, you can use the **Real-Time Workshop system code** field to control the format of the generated code.

- 4** Ensure that **Real-Time Workshop system code** is **Function** or **Reusable function**. Otherwise, you cannot specify memory sections for the subsystem.
- 5** If the code format is **Function** and you want separate data, check **Function with separate data**.

The **Real-Time Workshop** pane now shows all applicable memory section options. The available options depend on the values of **Real-Time Workshop system code** and the **Function with separate data** check box. When the former is **Function** and the latter is checked, the pane looks like this:



- 6 In the pull-down for each available definition category, specify the memory section (if any) that you want to apply to that category.
 - Selecting `Inherit from model` inherits the corresponding selection (if any) from the model level (not any parent subsystem).
 - Selecting `Default` specifies that the category has no associated memory section, overriding any model-level specification for that category.
- 7 Click **Apply** to save changes, or **OK** to save changes and close the dialog box.

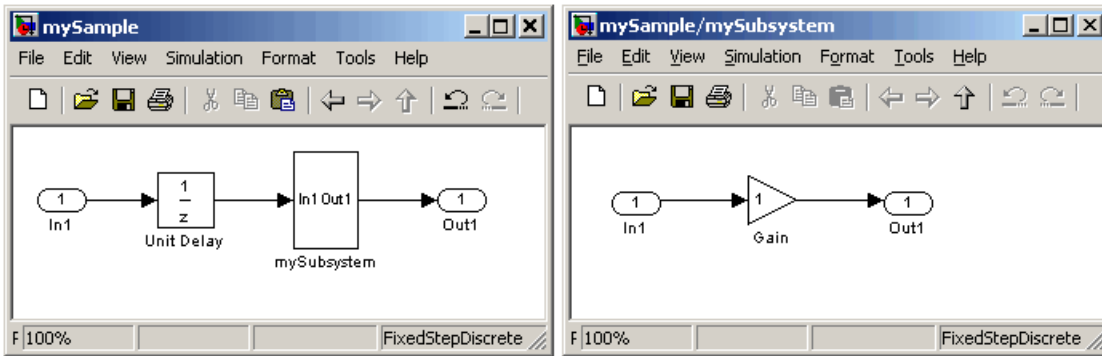
Caution If you use **Build Subsystem** to generate code for an atomic subsystem that specifies memory sections, the code generator ignores the subsystem-level specifications and uses the model-level specifications instead. The generated code is the same as if the atomic subsystem specified **Inherit from model** for every category of definition. For information about **Build Subsystem**, see “Generating Code and Executables from Subsystems”.

It is not possible to specify the memory section for a subsystem in a library. However, you can specify the memory section for the subsystem after you have copied it into a Simulink model. This is because in the library it is unknown what code generation target will be used. You can copy a library block into many different models with different code generation targets and different memory sections available.

Examples of Generated Code with Memory Sections

Sample ERT-Based Model with Subsystem

The next figure shows an ERT-based Simulink model that defines one subsystem, and the contents of that subsystem.



Assume that the subsystem is atomic, the **Real-Time Workshop system code** is Reusable function, memory sections have been created and assigned as shown in the next two tables, and all data memory sections specify **Pragma surrounds** to be Each variable.

Model-Level Memory Section Assignments and Definitions

Section Assignment	Section Name	Field Name	Field Value
Input/Output	MemSect1	Prepragma	#pragma IO_begin
		Postpragma	#pragma IO-end
Internal data	MemSect2	Prepragma	#pragma InData-begin(%<identifier>)
		Postpragma	#pragma InData-end
Parameters	MemSect3	Prepragma	#pragma Parameters-begin
		Postpragma	#pragma Parameters-end

Model-Level Memory Section Assignments and Definitions (Continued)

Section Assignment	Section Name	Field Name	Field Value
Initialize/ Terminate	MemSect4	Prepragma	#pragma InitTerminate-begin
		Postpragma	#pragma InitTerminate-end
Execution functions	MemSect5	Prepragma	#pragma ExecFunc-begin(%<identifier>)
		Postpragma	#pragma ExecFunc-begin(%<identifier>)

Subsystem-Level Memory Section Assignments and Definitions

Section Assignment	Section Name	Field Name	Field Value
Execution functions	MemSect6	Prepragma	#pragma DATA_SEC(%<identifier>, "FAST_RAM")
		Postpragma	

Given the preceding specifications and definitions, the code generator would create the following code, with minor variations depending on the current version of the Target Language Compiler.

Model-Level Data Structures

```
#pragma IO-begin
ExternalInputs_mySample mySample_U;
#pragma IO-end

#pragma IO-begin
ExternalOutputs_mySample mySample_Y;
#pragma IO-end

#pragma InData-begin(mySample_B)
BlockIO_mySample mySample_B;
#pragma InData-end

#pragma InData-begin(mySample_DWork)
D_Work_mySample mySample_DWork;
#pragma InData-end

#pragma InData-begin(mySample_M_)
RT_MODEL_mySample mySample_M_;
#pragma InData-end

#pragma InData-begin(mySample_M)
RT_MODEL_mySample *mySample_M = &mySample_M_;
#pragma InData-end

#pragma Parameters-begin
Parameters_mySample mySample_P = {
    0.0 , {2.3}
};
#pragma Parameters-end
```

Model-Level Functions

```
#pragma ExecFunc-begin(mySample_step)
void mySample_step(void)
{
    real_T rtb_UnitDelay;
    rtb_UnitDelay = mySample_DWork.UnitDelay_DSTATE;
    mySubsystem(rtb_UnitDelay, &mySample_B.mySubsystem;
```

```

        (rtP_mySubsystem *) &mySample_P.mySubsystem);
    mySample_Y.Out1_o = mySample_B.mySubsystem.Gain;
    mySample_DWork.UnitDelay_DSTATE = mySample_U.In1;
}
#pragma ExecFunc-end(mySample_step)

#pragma InitTerminate-begin
void mySample_initialize(void)
{
    rtmSetErrorStatus(mySample_M, (const char_T *)0);
    {
        ((real_T*)&mySample_B.mySubsystem.Gain)[0] = 0.0;
    }
    mySample_DWork.UnitDelay_DSTATE = 0.0;
    mySample_U.In1 = 0.0;
    mySample_Y.Out1_o = 0.0;
    mySample_DWork.UnitDelay_DSTATE = mySample_P.UnitDelay_X0;
}
#pragma InitTerminate-end

```

Subsystem Function

Because the subsystem specifies a memory section for execution functions that overrides that of the parent model, subsystem code looks like this:

```

/* File: mySubsystem.c */

#pragma DATA_SEC(mySubsystem, FAST_RAM )
void mySubsystem(real_T rtu_In1,
rtB_mySubsystem *localB,
rtP_mySubsystem *localP)
{
    localB->Gain = rtu_In1 * localP->Gain_Gain;
}

```

If the subsystem had not defined its own memory section for execution functions, but inherited that of the parent model, the subsystem code would have looked like this:

```

/* File: mySubsystem.c */

```

```
#pragma ExecFunc-begin(mySubsystem)
void mySubsystem(real_T rtu_In1,
rtB_mySubsystem *localB,
rtP_mySubsystem *localP)
{
    localB->Gain = rtu_In1 * localP->Gain_Gain;
}
#pragma ExecFunc-end(mySubsystem)
```

Memory Section Limitation

Memory sections cannot be applied to shared utility functions, such as lookup table functions, data type conversion functions, and fixed-point functions. For information about shared utilities, see “Setting Up Runtime Logging to MAT-Files”, “Supporting Shared Utility Directories in the Build Process”, and “Supporting the Shared Utilities Folder” in the Real-Time Workshop documentation.

Optimizing Buses for Code Generation

- “Introduction” on page 9-2
- “Setting Bus Diagnostics” on page 9-3
- “Optimizing Virtual and Nonvirtual Buses” on page 9-4
- “Using Single-Rate and Multi-Rate Buses” on page 9-7
- “Setting Bus Signal Initial Values” on page 9-11
- “Buses and Atomic Subsystems” on page 9-16

Introduction

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

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

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

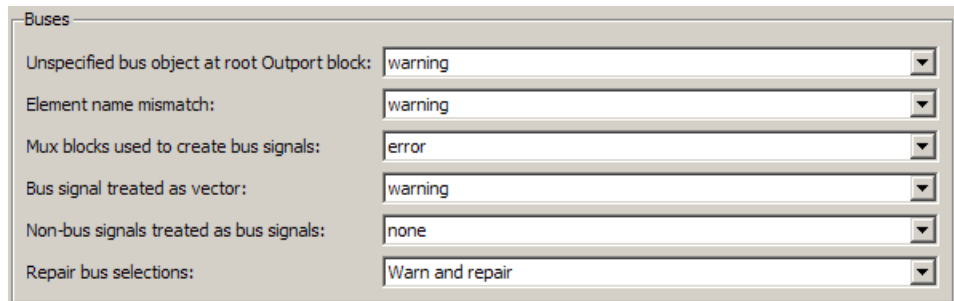
- Simplifying the layout of the model
- Increasing the efficiency of generated code
- Defining data structures for function/subsystem interfaces
- Defining data structures that match existing data structures in external C code

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

This chapter focuses on optimizations that are appropriate for final production code. Before you read this chapter, be sure that you have read “Using Composite Signals”. This chapter assumes that you understand all the concepts and procedures described in that one, including the blocks used for creating and manipulating buses.

Setting Bus Diagnostics

Simulink provides diagnostics that you can use to optimize bus usage. The MathWorks recommends setting the following values on the **Configuration parameters > Diagnostics > Connectivity** pane:



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

The last diagnostic, **Bus signal treated as vector**, is enabled only when **Mux blocks used to create bus signals** is set to error. Setting **Mux blocks used to create bus signals** to none disables both diagnostics. Temporarily disabling the two mux and bus diagnostics allows you to debug other bus problems before addressing mux and bus mixtures. You can then enable the last two diagnostics and use them to eliminate any such mixtures. When you build existing models, the diagnostic settings should be as shown at all times. See “Avoiding Mux/Bus Mixtures” for more information.

Optimizing Virtual and Nonvirtual Buses

In this section...
“Use Virtual Buses Wherever Possible” on page 9-4
“Avoid Nonlocal Nested Buses in Nonvirtual Buses” on page 9-5

Use Virtual Buses Wherever Possible

Virtual buses are graphical conveniences that do not affect generated code. As a result, the code generation engine is able to fully optimize the signals in the bus. You should therefore use virtual rather than nonvirtual buses wherever possible. You can convert between virtual and nonvirtual buses as needed using Signal Conversion blocks. In many cases, Simulink automatically converts a virtual bus to a nonvirtual bus when required. For example, a virtual bus input to a Model block becomes a nonvirtual bus with no need for explicit conversion. See for more information.

When are Virtual and Nonvirtual Buses Required?

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

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

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

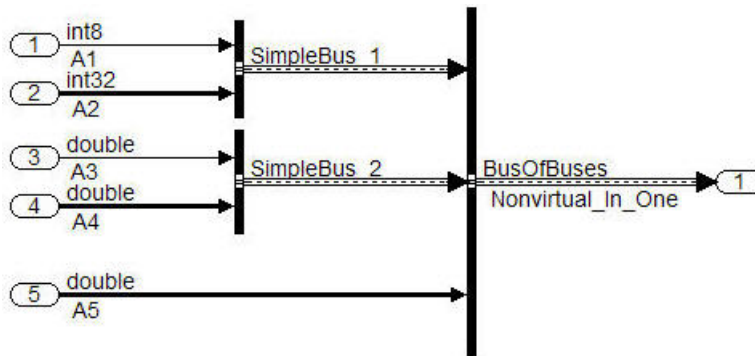
- Only virtual buses can be used for bundling function call signals.

Avoid Nonlocal Nested Buses in Nonvirtual Buses

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

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

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

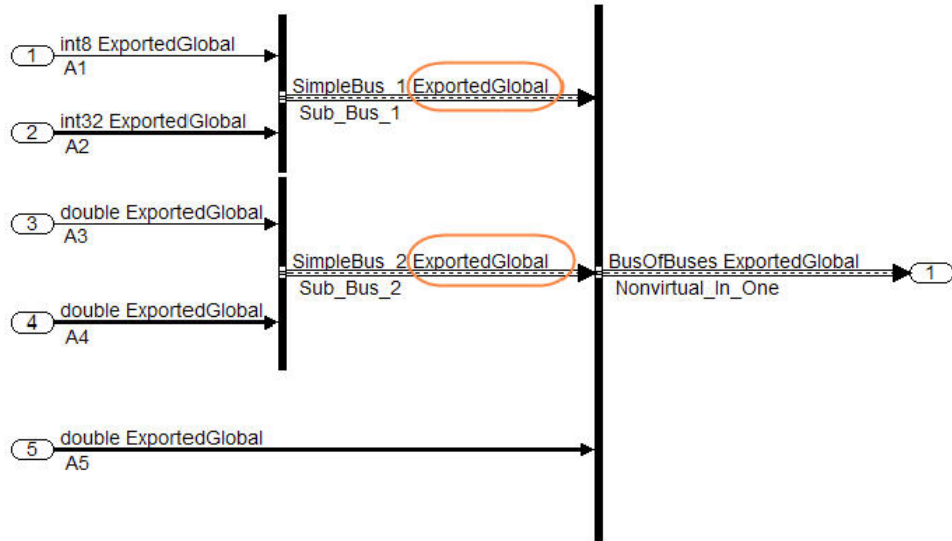


```

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

```

By contrast in the next example the subordinate elements sub_bus_1 and sub_bus_2 are global in scope. First the assignment to the subordinate bus occurs (lines 54 – 59) then the copy of the subordinate bus to the main bus (lines 60 – 61). In most cases, this is not an efficient implementation:



```

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

```

Using Single-Rate and Multi-Rate Buses

In this section...

“Introduction” on page 9-7

“Techniques for Combining Multiple Rates” on page 9-7

“Larger Buses and Multiple Rates” on page 9-9

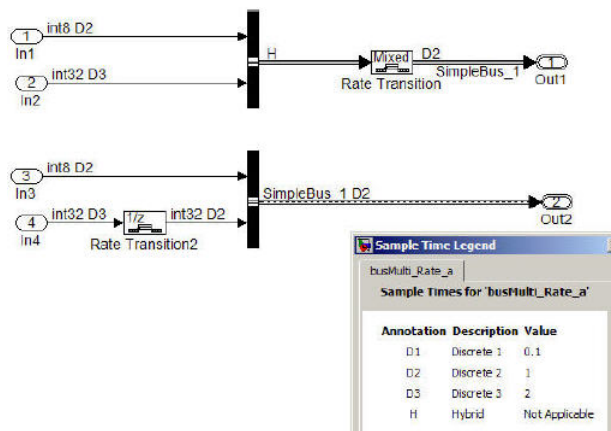
“Specifying Sample Time Rates” on page 9-10

Introduction

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

Techniques for Combining Multiple Rates

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

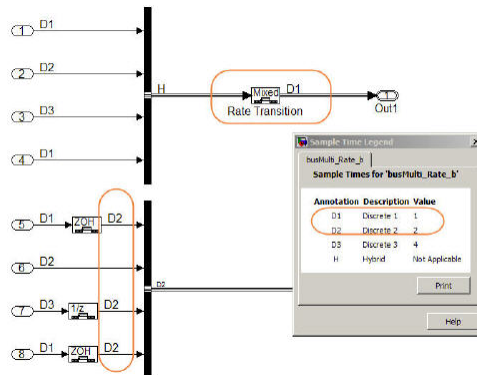


The signals with different rates in the first example can be combined into a virtual bus, because virtual buses support multiple rates. However, a multirate virtual bus cannot connect to a root-level output port. The bus therefore passes through a Rate Transition block that converts it to a single-rate bus, then connects to the Output. This technique is preferable only for virtual buses that contain no more than two signals. See “Larger Buses and Multiple Rates” on page 9-9.

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

Larger Buses and Multiple Rates

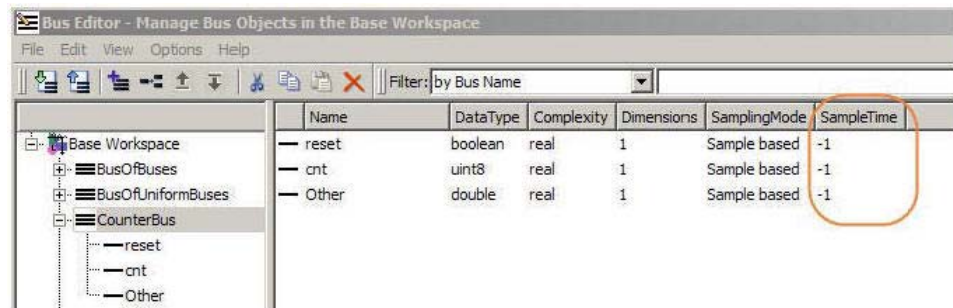
When you create a multirate virtual bus that contains more than two signals, you can convert the bus to single-rate by applying a Rate Transition block to the output of the Bus Creator block. However, The MathWorks recommends using a Rate Transition block on each input signal to give full control over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



Note that the preferred techniques for a virtual bus with more than two signals, and the required technique for a nonvirtual bus with any number of signals, are the same. Note also that, in the preceding figure, the blocks that perform rate transition are not actual Rate Transition blocks, but other blocks that can change the signal rate as part of some other operation. The identity of the blocks that perform rate transition makes no difference. All that matters is that the signal rates match when required.

Specifying Sample Time Rates

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



Setting Bus Signal Initial Values

In this section...

“Introduction” on page 9-11

“Initializing Bus Signals in Simulink” on page 9-11

“Bus Initialization in Stateflow and Embedded MATLAB” on page 9-12

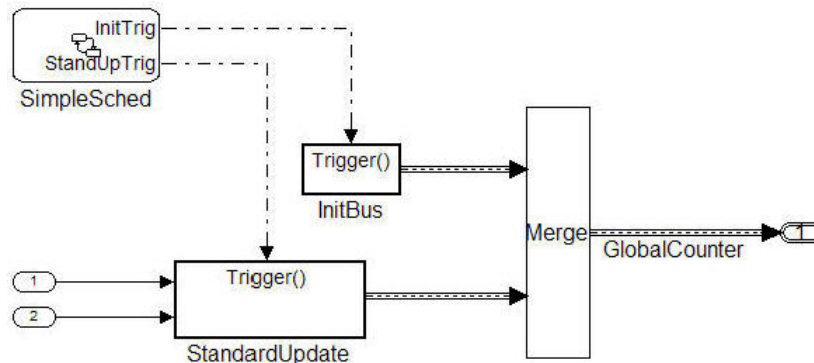
“Creating a Bus of Constants” on page 9-14

Introduction

Unlike scalar and vector signals, buses provide no direct way to initialize signals. This section describes techniques for initializing bus signals in Simulink, Stateflow, and Embedded MATLAB.

Initializing Bus Signals in Simulink

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

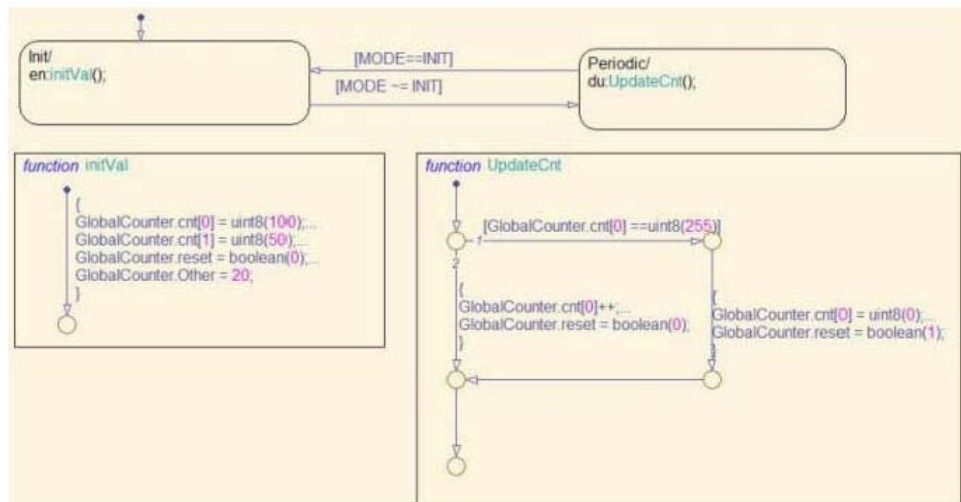


Both subsystems (`InitBus` and `StandardUpdate`) create a bus signal of type `CounterBus`. However, the assignment to the variable `GlobalCounter` is controlled by the Merge block. See “Function-Call Subsystems” for more information.

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

Bus Initialization in Stateflow and Embedded MATLAB

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

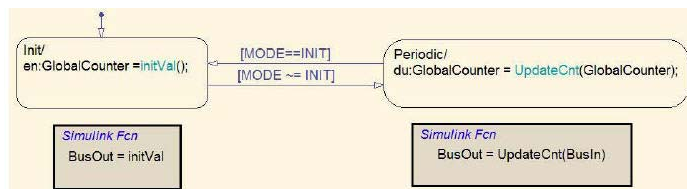


In the generated code, you can see that the UpdateCnt function uses the past value of GlobalCounter.cnt:

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

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

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



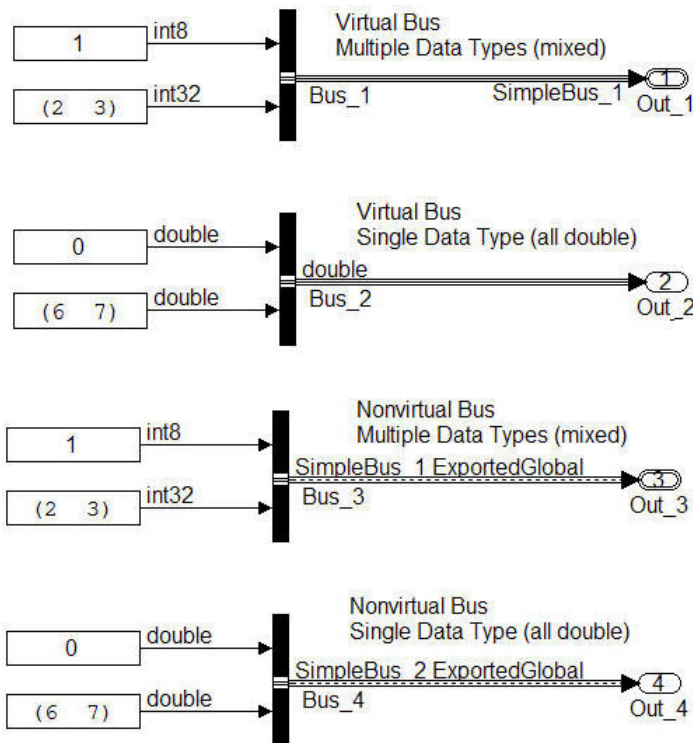
The Simulink subsystems are the same as those used in the earlier Simulink-only example.

Creating a Bus of Constants

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

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

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



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

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

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

To avoid repeatedly updating a bus of constants, place the bus code into a function-call subsystem, as described in “Initializing Bus Signals in Simulink” on page 9-11. When you use this technique, make sure the function-call subsystem is called at the start of execution. See “Function-Call Subsystems” for more information.

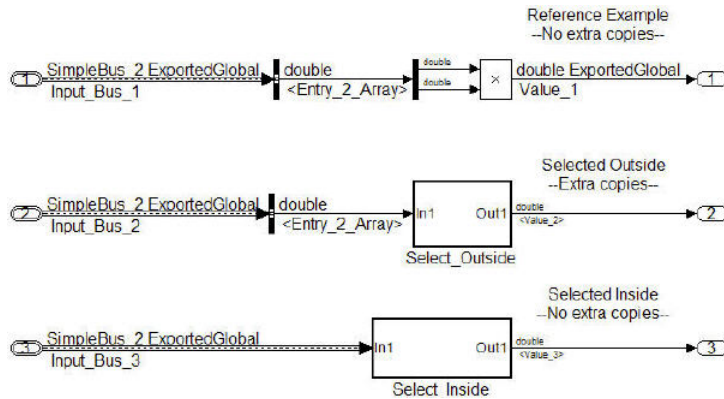
Buses and Atomic Subsystems

In this section...

- “Extract Nonvirtual Bus Signals Inside of Atomic Subsystems” on page 9-16
- “Virtual Bus Signals Crossing Atomic Boundaries” on page 9-17
- “Atomic Subsystems and Buses of Constants” on page 9-19

Extract Nonvirtual Bus Signals Inside of Atomic Subsystems

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



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

```

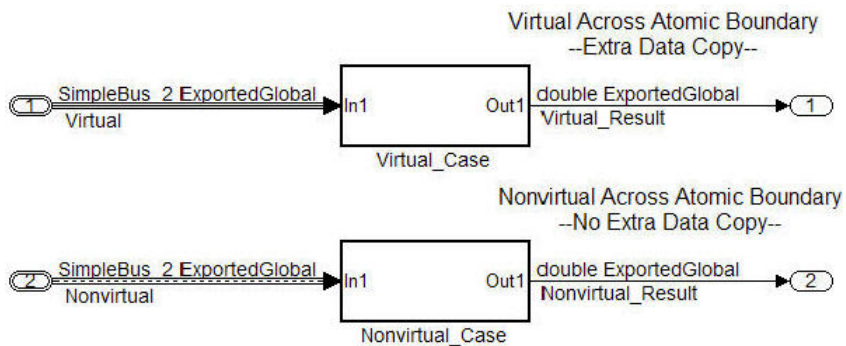
12 void virtualAcrossBo_Nonvirtual_Case(void)
13 {
14     Nonvirtual_Result = Nonvirtual.Entry_2_Array[0] * Nonvirtual.Entry_2_Array[1];
15 }
16
17 void virtualAcrossBound_Virtual_Case(void)
18 {
19     Virtual_Result = virtualAcrossBoundary_B.Entry_2_Array[0] *
20     virtualAcrossBoundary_B.Entry_2_Array[1];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

```

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

Virtual Bus Signals Crossing Atomic Boundaries

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

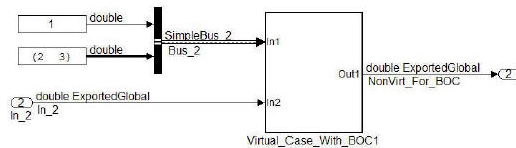
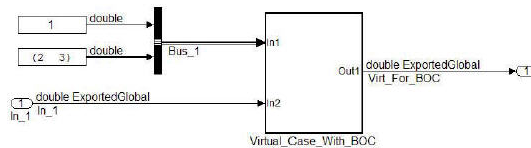


```
12 void virtualAcrossBo_Nonvirtual_Case(void)
13 {
14     Nonvirtual_Result = Nonvirtual.Entry_2_Array[0] * Nonvirtual.Entry_2_Array[1];
15 }
16
17 void virtualAcrossBound_Virtual_Case(void)
18 {
19     Virtual_Result = virtualAcrossBoundary_B.Entry_2_Array[0] *
20     virtualAcrossBoundary_B.Entry_2_Array[1];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

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

Atomic Subsystems and Buses of Constants

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



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

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

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


Renaming and Replacing Data Types

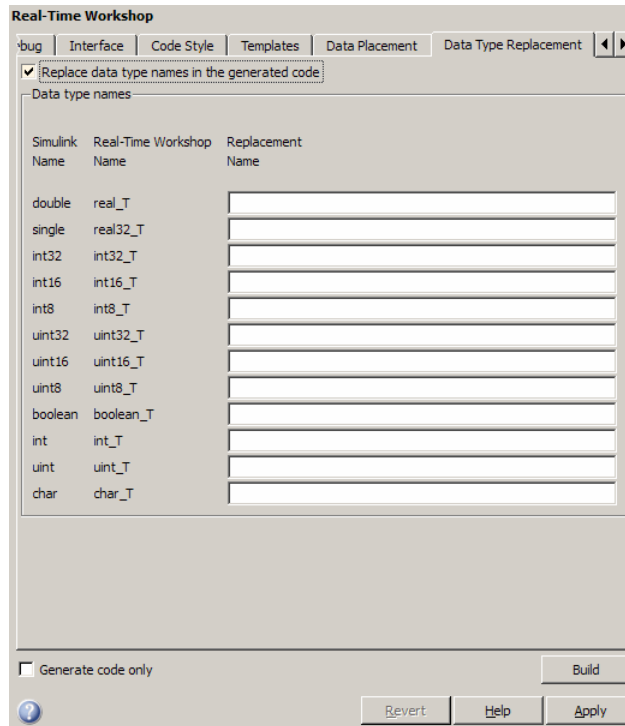
- “Defining Application-Specific Data Types Based On Built-In Types” on page 10-2
- “Code Generation with User-Defined Data Types” on page 10-4

Defining Application-Specific Data Types Based On Built-In Types

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

To configure replacement data types,

- 1 Click **Replace data type names in the generated code**. A **Data type names** table appears. The table lists each Simulink built-in data type name with its corresponding Real-Time Workshop data type name.



- 2 Selectively fill in fields in the third column with your replacement data types. Each replacement data type should be the name of

a `Simulink.AliasType` object that exists in the base workspace. Replacements may be specified or not for each individual built-in type.

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

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

An error occurs if a replacement data type specification is inconsistent. For more information, see “Replacing Built-In Data Type Names in Generated Code” on page 11-52.

Code Generation with User-Defined Data Types

In this section...
“Overview” on page 10-4
“Specifying Type Definition Location for User-Defined Data Types” on page 10-5
“Using User-Defined Data Types for Code Generation” on page 10-6

Overview

The Real-Time Workshop Embedded Coder software supports use of user-defined data type objects in code generation. These include objects of the following classes:

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

For information on the properties and usage of these data object classes, see `Simulink.AliasType`, `Simulink.Bus`, `Simulink.NumericType`, and `Simulink.StructType` in the “Simulink Classes” section of the Simulink Reference documentation. For general information on creating and using data objects, see the “Working with Data Objects” section of the Simulink documentation

In code generation, you can use user-defined data objects to

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

In general, code generated from user-defined data objects conforms to the properties and attributes of the objects as defined for use in simulation.

When generating code from user-defined data objects, the name of the object is the name of the data type that is used in the generated code. Exception: for `Simulink.NumericType` objects whose `IsAlias` property is false, the name of the functionally equivalent built-in or fixed-point Simulink data type is used instead.

Note The names of data types defined using `Simulink.AliasType` objects are preserved in the generated code only for installations with a Real-Time Workshop Embedded Coder license.

Specifying Type Definition Location for User-Defined Data Types

When a model uses `Simulink.DataType` and `Simulink.Bus` objects, corresponding typedefs are needed in code. Both `Simulink.DataType` and `Simulink.Bus` objects have a `HeaderFile` property that controls the location of the object's typedef. Setting a `HeaderFile` is optional and affects code generation only.

Omitting a HeaderFile Value

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

Example. For a `Simulink.NumericType` object named `myfloat` with a `Category` of `double` and no `HeaderFile` property specified, `model_types.h` in the generated code contains:

```
typedef real_T myfloat;
```

Specifying a HeaderFile Value

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

- The string must be the name of a header file that contains a typedef for the object.

- The generated file `model_types.h` contains a `#include` that gives the header file name.

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

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

This example:

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

generates the directive:

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

This example:

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

generates the directive:

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

Using User-Defined Data Types for Code Generation

To specify and use user-defined data types for code generation:

- 1 Create a user-defined data object and configure its properties, as described in the “Working with Data Objects” section of the Simulink documentation.
- 2 If you specified the `HeaderFile` property, copy the header file to the appropriate directory.
- 3 Set the output data type of selected blocks in your model to the user-defined data object. To do this, set the **Data type** parameter of the block to `Specify with dialog`. Then, enter the object name in the **Output data type** parameter.
- 4 The specified output data type propagates through the model and variables of the user-defined type are declared as required in the generated code.

Managing Data Definitions and Declarations With the Data Dictionary

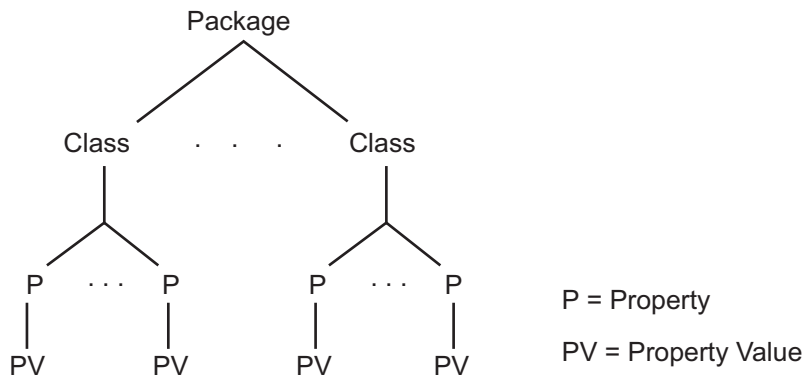
- “Overview of the Data Dictionary” on page 11-2
- “Creating Simulink and mpt Data Objects” on page 11-4
- “Creating a Data Dictionary for a Model” on page 11-19
- “Defining All Global Data Objects in a Separate File” on page 11-26
- “Defining a Specific Global Data Object in Its Own File” on page 11-28
- “Saving and Loading Data Objects” on page 11-29
- “Applying Naming Rules to Identifiers Globally” on page 11-30
- “Creating User Data Types” on page 11-38
- “Selecting User Data Types for Signals and Parameters” on page 11-43
- “Registering mpt User Object Types” on page 11-48
- “Replacing Built-In Data Type Names in Generated Code” on page 11-52
- “Customizing Data Object Wizard User Packages” on page 11-61

Overview of the Data Dictionary

A data dictionary contains all of the parameters and signals that the source code uses, and a description of their properties. The data dictionary that is created for Simulink and Stateflow models is called the code generation data dictionary. (You can use the data dictionary for simulation. This does not require that you have a Real-Time Workshop Embedded Coder license.) The dictionary is the total number of data objects that appear in the middle pane of the Model Explorer. These data objects also appear in the MATLAB workspace. The procedure described in this chapter allows you to create or edit the dictionary. The procedure allows you to control property values for each data object. This, in turn, determines how each parameter and signal is defined and declared in the automatically generated code.

The values of data object properties can affect where the code generator places a parameter or signal in the generated file. This is because some property values are associated with different template symbols. The location of a symbol in a template determines where the associated parameter or signal is located in the generated file. For details about templates and symbols, see “Configuring Templates for Customizing Code Organization and Format” on page 16-22.

It is helpful to define terms you will see when managing the dictionary, especially when you view them using the Model Explorer. The Simulink software uses a hierarchy of terms that are drawn from object-oriented programming. For details, see “Working with Data Objects” in the Simulink documentation. The sketch below summarizes this hierarchy.



Simulink or mpt is the package. Parameter and Signal are two classes in each of these packages. Each class has a number of properties associated with it. Sometimes properties are called *attributes*. Data objects (the parameters and signals) are the instances of a `package.class` that make up the data dictionary. All parameter data objects have a set of properties. All signal data objects have a different set of properties than that for parameters. For each data object, each property in the set has its own property *value* that must be specified in the dictionary.

Creating Simulink and mpt Data Objects

In this section...

“Overview” on page 11-4

“Creating Simulink Data Objects with Data Object Wizard” on page 11-5

“Creating mpt Data Objects with Data Object Wizard” on page 11-11

“Comparing Simulink and mpt Data Objects” on page 11-12

“Creating Data Objects Based on an External Data Dictionary” on page 11-16

Overview

The Real-Time Workshop Embedded Coder software provides the mpt (module packaging tool) data object, which contains all the properties of Simulink data objects plus properties that provide additional control over module packaging. For a comparison of the properties of Simulink and mpt data objects, see “Comparing Simulink and mpt Data Objects” on page 11-12.

There are different ways of creating Simulink and mpt data objects for a data dictionary.

- One-by-one, either using the MATLAB command line or using the Model Explorer **Add** menu and selecting **Simulink Parameter**, **Simulink Signal**, **MPT Parameter**, or **MPT Signal**. For more information, see “Working with Data Objects” in the Simulink documentation.
- All at once, invoking Data Object Wizard for an existing model. For more information and examples, see “Data Object Wizard” in the Simulink documentation and “Creating mpt Data Objects with Data Object Wizard” on page 11-11.
- Creating data objects based on an external data dictionary. You can do this manually item by item, or all at once automatically using a script. For more information, see “Creating Data Objects Based on an External Data Dictionary” on page 11-16.

The following sections illustrate how to create Simulink and mpt data objects and compares their properties as data types.

Creating Simulink Data Objects with Data Object Wizard

You can use Data Object Wizard to create data objects for your model (see “Data Object Wizard” in the Simulink documentation).

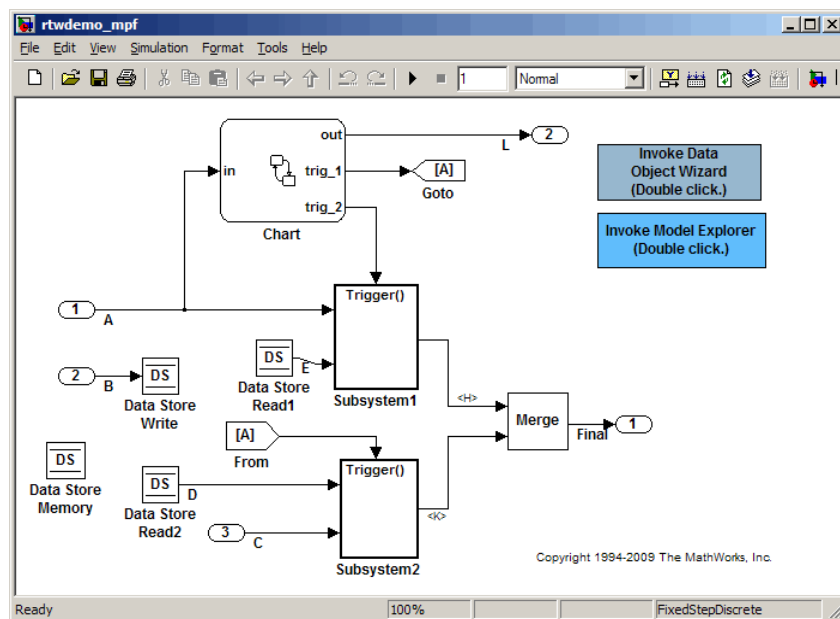
Data Object Wizard is especially useful for creating multiple data objects for

- Existing models that do not currently use data objects.
- Existing models to which you have added signals or parameters and therefore you need to create more data objects.

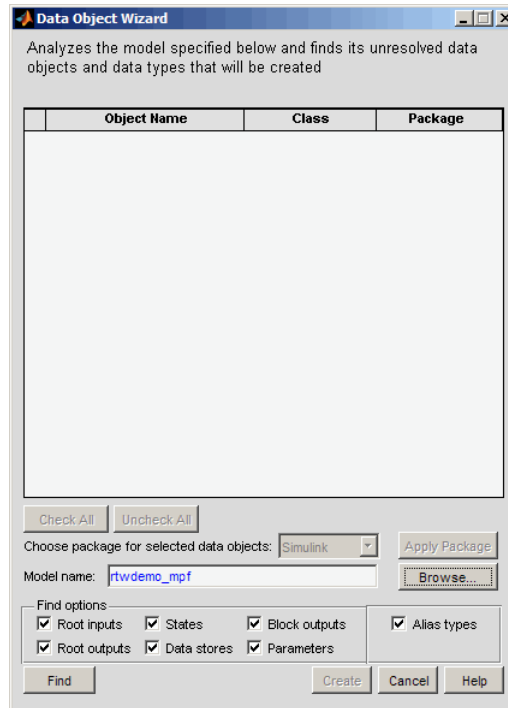
Creating Simulink Data Objects

This procedure creates Simulink data objects using Data Object Wizard.

- 1 Open the model whose data objects you want to be in the data dictionary. For example, open `rtwdemo_mpf.mdl` (which is located in `toolbox/rtw/rtwdemos`). This model appears as shown below.



- Open Data Object Wizard by entering `dataobjectwizard` at the MATLAB command line or by selecting **Data Object Wizard** from the **Tools** menu of your model. The Data Object Wizard dialog box appears:



The **Model name** field displays the name of the model. You could specify a different model by editing the field or by selecting the model using the adjacent **Browse** button. When the **Model name** field is nonempty, the **Find** button is enabled.

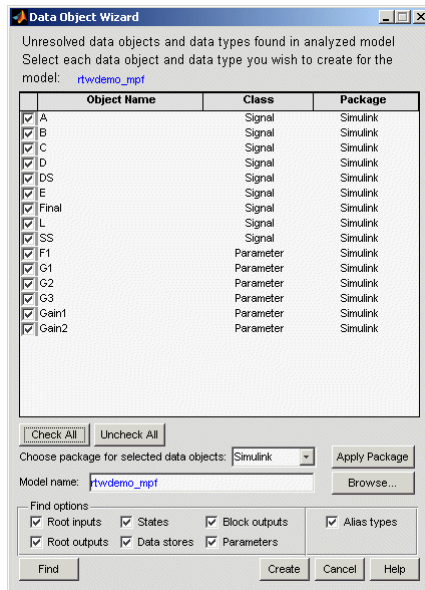
- In the **Find options** pane, select the desired check boxes. For descriptions of each check box, see “Data Object Wizard” in the Simulink documentation. Be sure to check the **Alias types** option. This finds all user-registered data types in the `sl_customization.m` file plus all data type replacements specified for the model in the **Data Type Replacement** pane of the Configuration Parameters dialog box. Data Object Wizard can create `Simulink.AliasType` objects from these.

4 Click the **Find** button. After a moment, a list of all of the model's potential data objects appear that are not yet in the code generation data dictionary, as shown below. This includes all of the model's signals (root inputs, root outputs, and block outputs), discrete states, data stores, and parameters, depending on:

- The check boxes you selected in the previous step
- The constraint mentioned in the note above

Data Object Wizard finds only those signals, parameters, data stores, and states whose storage class is set to Auto. The Wizard lists each data store and discrete state that it finds as a signal class.

5 Click **Check All** to select all data objects. Notice in the **Choose package for selected data objects** field that **Simulink**, the default, is selected. So all of the data objects are associated with the **Simulink** package, as shown below.



6 Click **Create**. The data objects are added to the MATLAB workspace, and they disappear from Data Object Wizard.

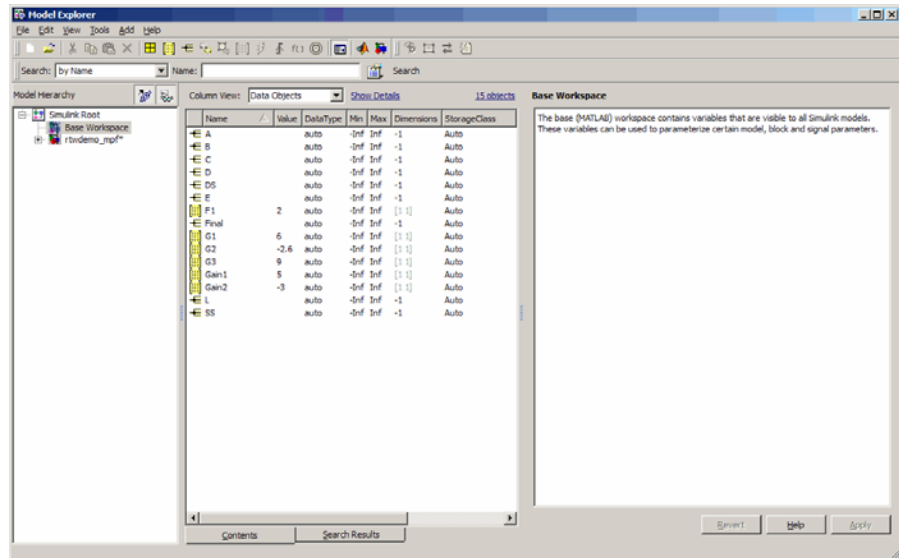
7 Click **Cancel**. The Data Object Wizard dialog box disappears.

Now you can set property values for the data objects.

Setting Property Values for Simulink Data Objects

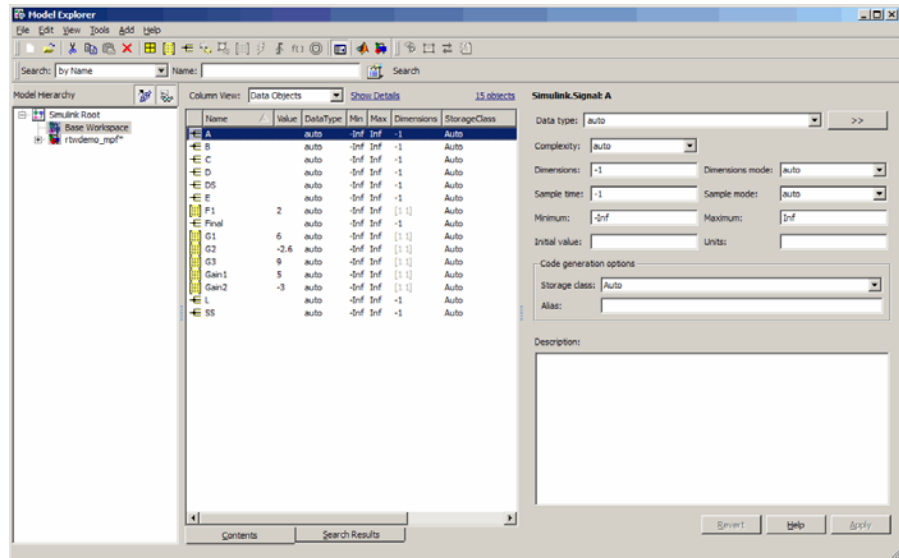
Most of the property values of data objects are supplied by defaults. A few are from the model. Note that for Simulink data objects, the default storage class is Auto.

- 1 Type `daexplr` on the MATLAB command line, and press **Enter**. The Model Explorer appears.
- 2 In the Model Hierarchy (left) pane, select **Base Workspace**. All of the Simulink data objects in the code generation data dictionary appear in the **Contents** (middle) pane, as shown below.

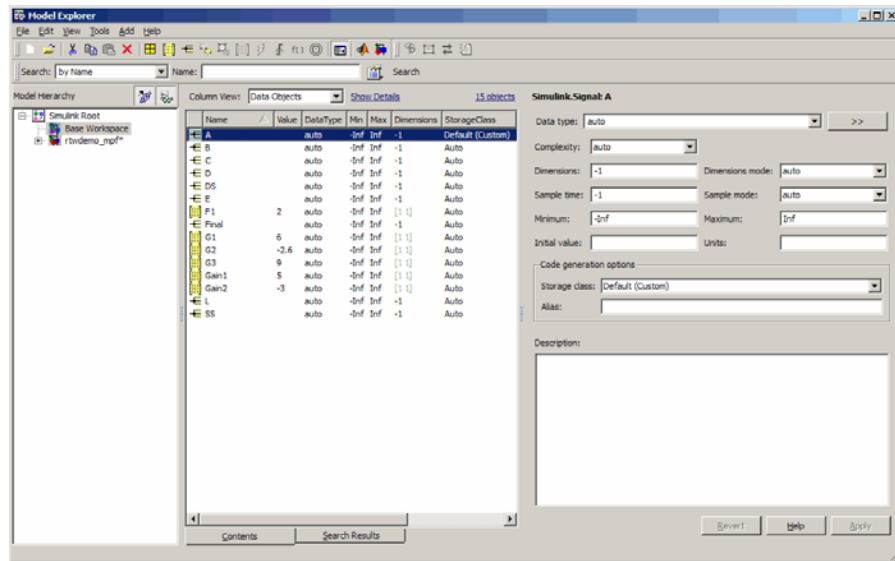


If the objects that you see do not appear in the order shown, click the **Name** column header in the middle pane to sort the objects in ascending order by name.

- To see the properties of a Simulink data object, select a data object in the middle pane. The right pane displays the property names for that object. These property names also appear as column headings in the middle pane. Reshape the middle pane as needed to see all the headings. For example, if you select signal data object A in the middle pane, the Model Explorer looks like this:



- 4 You can change the values specified for the properties of the selected object. For example, with A selected, change its **StorageClass** property from Auto to Default (Custom), then click **Apply**. The property changes as shown below:



You can use Control-Right-Click to select multiple objects in the center pane, then edit any property value. The wizard applies the new value to all selected objects. For descriptions of object properties and their values, see Parameter and Signal Property Values on page 6-2.

Generating and Inspecting Code

All data objects for the model are in the code generation data dictionary. You have specified property values for each data object's properties as needed. Now you generate and inspect the source code, to see if it needs correction or modification. If it does, you can change property values and regenerate the code until it is what you want.

- 1 In the Configuration Parameters dialog box, click **Real-Time Workshop** in the left pane.
- 2 In the **Report** pane, select the **Create code generation report** check box.

Note When you select the **Create code generation report** check box, the Real-Time Workshop Embedded Coder software automatically selects two check boxes on the pane: **Launch report automatically** and **Code-to-model**. For large models, you may find that HTML report generation (step 4 below) takes longer than you want. In this case, consider clearing the **Code-to-model** check box (and the **Model-to-code** check box if selected). The report will be generated faster.

- 3** In the **Real-Time Workshop** pane, select the **Generate code only** check box. The **Build** button changes to **Generate code**.

Note The generate code process generates the `.c/ .cpp` and `.h` files. The build process adds compiling and linking to generate the executable. For details on build, see “Understanding the Build Process” in the Real-Time Workshop documentation.

- 4** Click the **Generate code** button. After a moment, the HTML code generation report appears, listing the generated files on the left pane.
- 5** Select and review files in the HTML code generation report. See “Generating an HTML Code Generation Report” on page 19-4 for more information.

Creating mpt Data Objects with Data Object Wizard

Create mpt data objects using Data Object Wizard the same way you did for Simulink data objects, as explained in “Creating Simulink Data Objects with Data Object Wizard” on page 11-5, except select mpt as the package instead of Simulink.

Set the property values for the mpt data objects the same way you set them for Simulink data objects, as explained in “Setting Property Values for Simulink Data Objects” on page 11-8, with the following exceptions:

- Accept the default custom storage class for mpt data objects, `Global(Custom)`

- For data objects A and F1, type `mydefinitionfile` in the **Definition file** field on the Model Explorer.

Then generate and inspect the code.

Note The **Alias** field is related to “Applying Naming Rules to Identifiers Globally” on page 11-30.

Comparing Simulink and mpt Data Objects

The `mpt` data object contains all the properties of Simulink data objects plus properties that provide additional control over module packaging. The differences between Simulink and `mpt` data objects can be illustrated by comparing

- “Signal and Parameter Properties” on page 11-13
- “Configuration Parameters” on page 11-14
- “Generated Code” on page 11-15

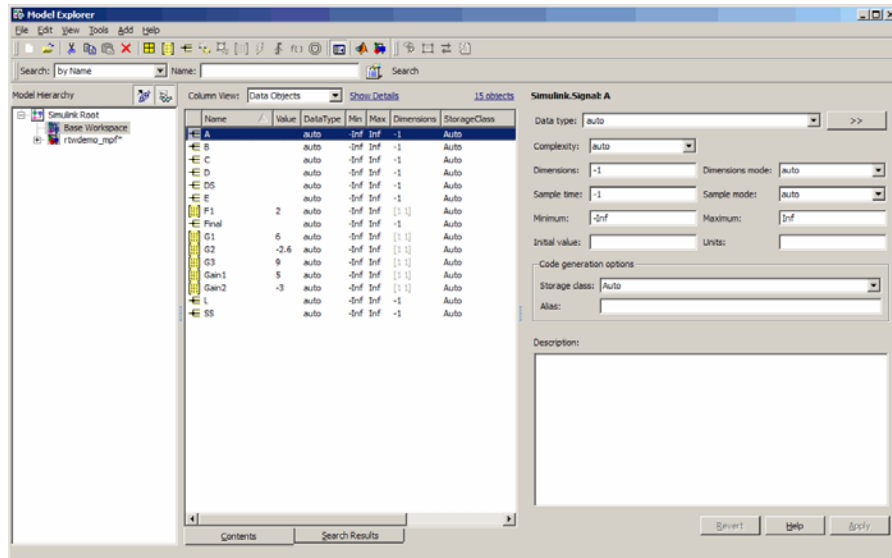
Key differences include the following:

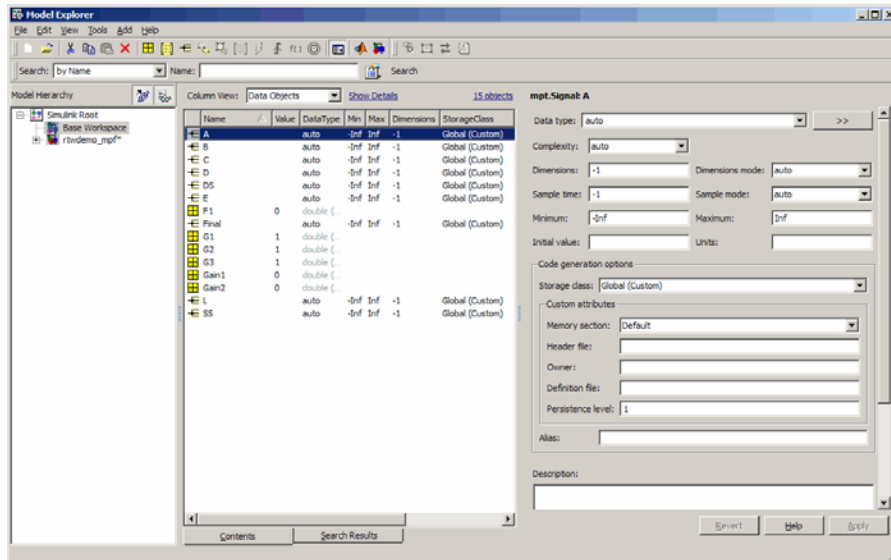
- Different custom storage classes displayed in the Model Explorer for `mpt` data objects provide more control over the appearance of the generated code.
- Additional custom attributes (owner, definition file, persistence level, memory section) for `mpt` data objects provide more control over data packaging in the generated code.
- On the **Comments** pane of the Configuration Parameters dialog box, the **Custom comments (MPT objects only)** option allows you to add a comment just above a signal or parameter’s identifier in the generated code.
- On the **Data Placement** pane of the Configuration Parameters dialog box, in the **Global data placement (MPT data objects only)** subpane:
 - The **Module naming** parameter allows you to name the module that owns the model
 - The **Signal display level** parameter allows you to specify whether or not the code generator declares a signal data object as global data

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

Signal and Parameter Properties

The properties that appear in Model Explorer when mpt is the package include all the properties that appear when Simulink is the package plus additional properties. Notice this by comparing the next two figures. (For descriptions of all properties in Model Explorer, see Parameter and Signal Property Values on page 6-2.)





Configuration Parameters

The following configuration parameters relate to Real-Time Workshop Embedded Coder module packaging features. These parameters are available in the Configuration Parameters dialog box and Model Explorer when the system target file selected for a Simulink model is `ert.tlc` (or a system target file derived from an `ert.tlc`):

- **Custom comments (MPT objects only)** option on the **Real-Time Workshop/Comments** pane
- In the **Global data placement (MPT data objects only)** subpane on the **Real-Time Workshop/Data Placement** pane:
 - **Module naming** parameter
 - **Signal display level** parameter
 - **Parameter tune level** parameter

Generated Code

In the example used in “Setting Property Values for Simulink Data Objects” on page 11-8, you selected Default (Custom) in the **Storage class** field for signal A and parameter F1. You selected the default Auto in the **Storage class** field for the remaining data objects. But for the mpt data objects you used the default Global (Custom) in the **Storage class** field for all data objects. When you generated code, these selections resulted in the definitions and declarations shown in the table below.

Simulink Data Object with Auto Storage Class	Simulink Data Object with Default (Custom) Storage Class	mpt Data Object with Global (Custom) Storage Class and Definition File Named mydefinitionfile
<pre>In rtwdemo_mpf.c: /* For signal A */ ExternalInputs rtU; /* For parameter F1 */ if(rtU.A * 2.0 > 10.0) {... In rtwdemo_mpf.h: /* For signal A */ typedef struct { real_T A; } ExternalInputs; extern ExternalInputs rtU;</pre>	<pre>In global.c: real_T A; real_T F1 = 2.0; In global.h: extern real_T A; extern real_T F1;</pre>	<pre>In mydefinitionfile.c: real_T A; real_T F1 = 2.0; In global.h: extern real_T A; extern real_T F1;</pre>

The results shown in the second and third columns of the preceding table require the following configuration parameter adjustments on the **Real-Time Workshop > Data Placement** pane:

- Set **Data definition** to Data defined in single separate source file.
- Set **Data definition filename** to global.c

- Set **Data declaration** to Data declared in single separate source file.
- Set **Data definition filename** to `global.h`

See the left column of the table, which shows generated code for Simulink signal and parameter data objects, whose **Storage class** field is `Auto`. The input `A` is defined as part of the structure `rtU` as shown above. In the case of the Simulink parameter data object `F1`, since the **StorageClass** was set to `auto`, the code generator chose to include the literal value of `F1` in the generated code. `F1` is a constant in the Stateflow diagram whose value is initialized as `2.0`:

```
if(rtU.A * 2.0 > 10.0) { ...
```

For more details, see “Introduction to Custom Storage Classes” on page 7-2 in the Real-Time Workshop Embedded Coder documentation and “Summary of Signal Storage Class Options” in the Real-Time Workshop documentation.

See the middle column of the table. The Simulink data objects whose **Storage class** is not `Auto` are defined in a definition statement in the global source file (`global.c`) and declared in a declaration statement in the global header file (`global.h`).

In the right column, Simulink data objects whose **Storage class** is not `Auto` are defined in `mydefinitionfile`, as you specified. The declarations for those objects are in the global header file.

Creating Data Objects Based on an External Data Dictionary

This procedure creates data objects based on an external data dictionary (such as a Microsoft® Excel® file). You can do this manually (that is, one-by-one) or automatically (all at once).

Manually Creating Objects to Represent External Data

You can create data objects (and their properties) one-by-one, based on an external data dictionary, as follows:

- 1** Open the external file that contains the data (such as a spreadsheet or database file).
- 2** Determine all of the data in this file that correspond to the parameters and signals in the model. In the code generation data dictionary, parameters in the external file belong to the Simulink parameter class and signals belong to the Simulink signal class.
- 3** On the MATLAB command line, type `daexplr` and press **Enter**. The Model Explorer appears.
- 4** On the **Model Hierarchy** (left) pane, expand **Simulink Root**, and select **Base Workspace**.
- 5** On the **Add** menu, select **MPT Parameter** or **Simulink Parameter**. The default name `Param` appears in the **Contents of** (middle) pane.
- 6** Double-click `Param` and rename this data object as desired.
- 7** Repeat steps 5 and 6 for each additional data item in the external file that belongs to the `mpt.Parameter` class or `Simulink.Parameter` class.

Now you will add data items in the external file that belong to the `mpt.Signal` class or `Simulink.Signal` class.

- 8** On the **Add** menu, select **MPT Signal** or **Simulink Signal**. The default name `Sig` appears in the **Contents of** pane.
- 9** Double-click `Sig` and rename the data object as desired.
- 10** Repeat steps 8 and 9 for each additional data item in the external file that belongs to the `mpt.Signal` class or `Simulink.Signal` class.

All external data items for the `mpt.Parameter` or `Simulink.Parameter` class, and the `mpt.Signal` or `Simulink.Signal` class now appear in the **Contents of** pane and in the MATLAB workspace. Therefore, they have been created in the code generation data dictionary.

Note The property *values* for these data objects are supplied by default.

Automatically Creating Objects to Represent External Data

You can create data objects (and their properties) all at once, based on an external data dictionary by creating and running a `.m` file. This file contains the same MATLAB commands you could use for creating data objects one-by-one on the command line, as explained in “Working with Data Objects” in the Simulink documentation. But instead of using the command line, you place the MATLAB commands in the `.m` file for *all* of the desired data in the external file:

- 1 Create a new `.m` file.
- 2 Place information in the file that describes all of the data in the external file that you want to be data objects. For example, the following information creates two `mpt` data objects with the indicated properties. The first is for a parameter and the second is for a signal:

```
% Parameters
mptParCon = mpt.Parameter;
mptParCon.RTWInfo.CustomStorageClass = 'Const';
mptParCon.value = 3;
% Signals
mptSigGlb = mpt.Signal;
mptSigGlb.DataType = 'int8';
```

- 3 Run the `.m` file. The data objects appear in the MATLAB workspace.

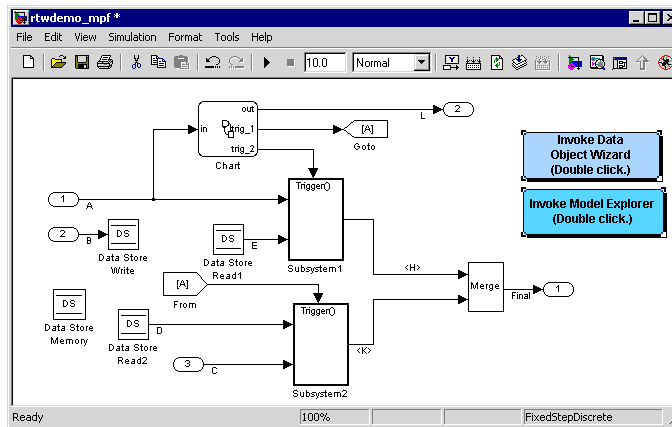
Note If you want to import data from an external data dictionary, you can write functions that read the information, convert these to data objects, and load them into the MATLAB workspace. Among available MATLAB functions that you can use for this process are `xmlread`, `xmlwrite`, `xlsread`, `xlswrite`, `csvread`, `csvwrite`, `dlmread`, and `dlmwrite`.

Creating a Data Dictionary for a Model

In this procedure, you create a data dictionary for a model using Data Object Wizard, inspect the data dictionary, and generate code. Definitions for the data objects in the dictionary are generated into the model source file (model.c).

Using Data Object Wizard

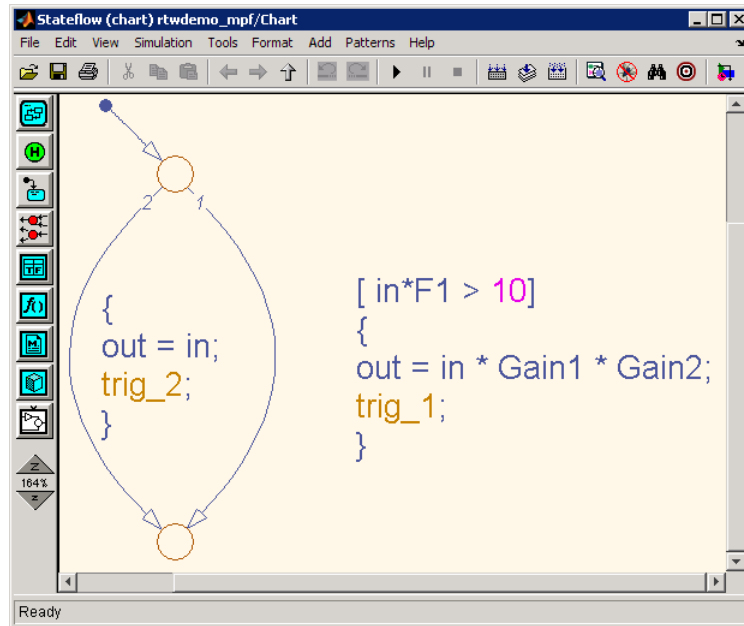
- 1 Open the demo model `rtwdemo_mpf` by clicking the link or by typing `rtwdemo_mpf` in the MATLAB Command Window.



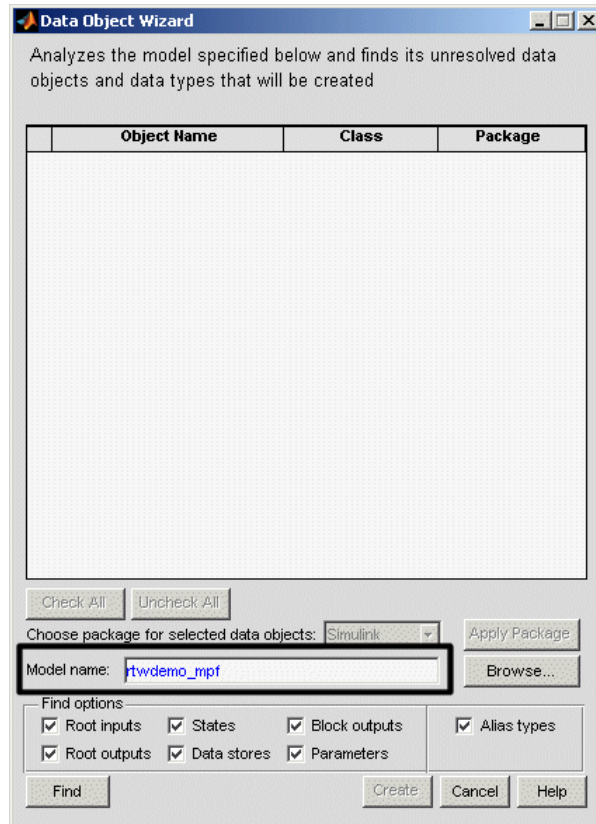
In this model,

- `A`, `B`, and `C` are input signals, and `L` and `Final` are output signals.
- `Subsystem1` receives inputs `A` and `E` and contains constants `G1` and `G2`. Signal `E` is an output from Data Store Read1.
- `Subsystem2` receives inputs `C` and `D`. Signal `D` is an output from Data Store Read2. There is a constant in `Subsystem2` named `G3`. Also, this subsystem has a Unit Delay block whose state name is `SS`.

- 2 Double-click the Stateflow chart and notice it has constants F1, Gain1, and Gain2, as shown below:

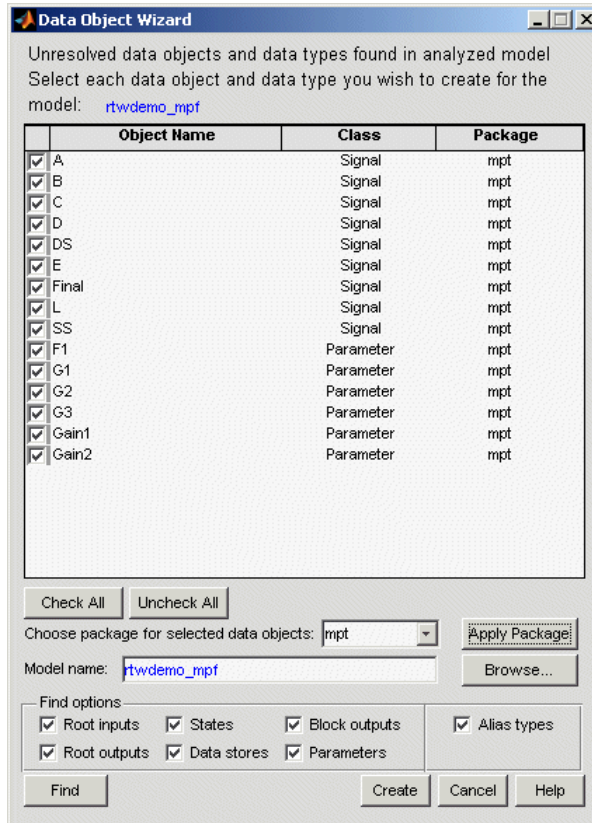


- 3 Change to a work directory that is not on an installation path and save the model in that work directory. The Real-Time Workshop software does not allow you to generate code from an installation directory.
- 4 Double-click the **Invoke Data Object Wizard** button on the model. Or, type `dataobjectwizard('rtwdemo_mpf')` in the MATLAB Command Window. Data Object Wizard opens and `rtwdemo_mpf` appears in the **Model name** field, as shown below.

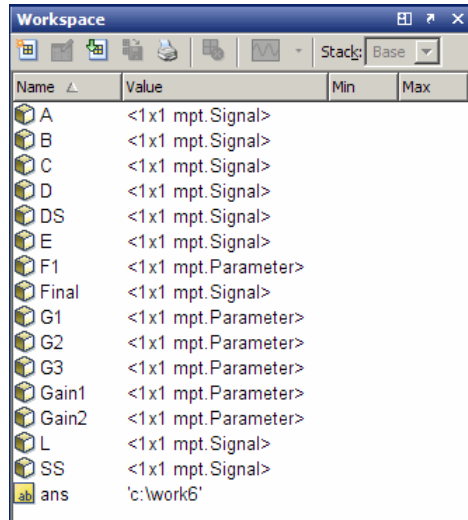


- 5** Click **Find** on Data Object Wizard. After a moment, the model's parameters and signals appear in Data Object Wizard. These "data objects" make up the data dictionary.
- 6** Click **Check All**, to select all data objects for the data dictionary.
- 7** In the **Choose package for selected objects** field, select `mpt`. For an explanation of "package," see "Overview of the Data Dictionary" on page 11-2.

- 8 Click Apply Package.** Data Object Wizard associates the selected data objects with the mpt package, as shown below.



- 9 Click **Create**. Data Object Wizard creates a data dictionary, consisting of data objects for the selected parameters and signals. Data Object Wizard removes the objects from its object view. Also, the objects are added to the MATLAB workspace, as shown below.



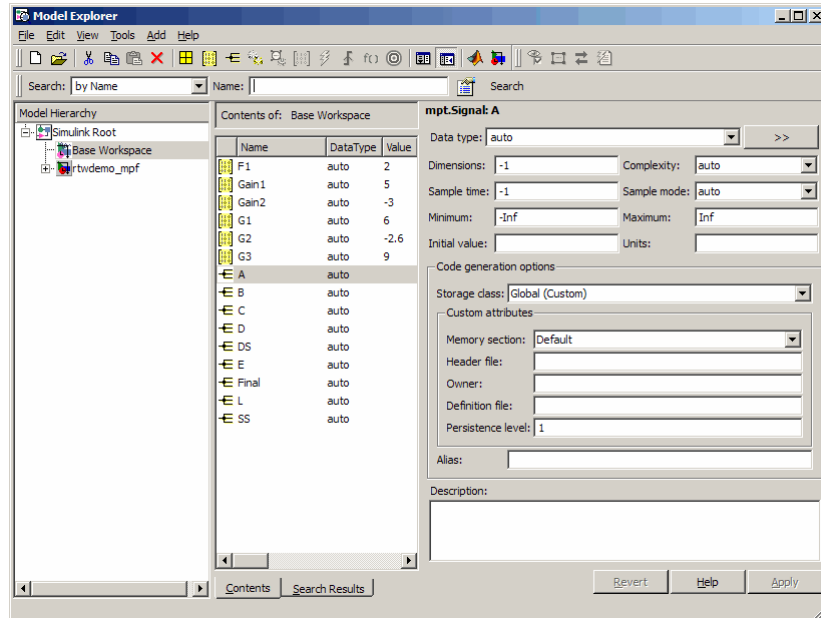
- 10 Close Data Object Wizard.

Inspect the Data Dictionary

You can verify that each data object you selected in Data Object Wizard is in the data dictionary, using the Model Explorer:

- 1 Open the Model Explorer.
- 2 In the left pane, select **Base Workspace**. Notice that all data objects that you placed in the data dictionary appear in the middle pane.

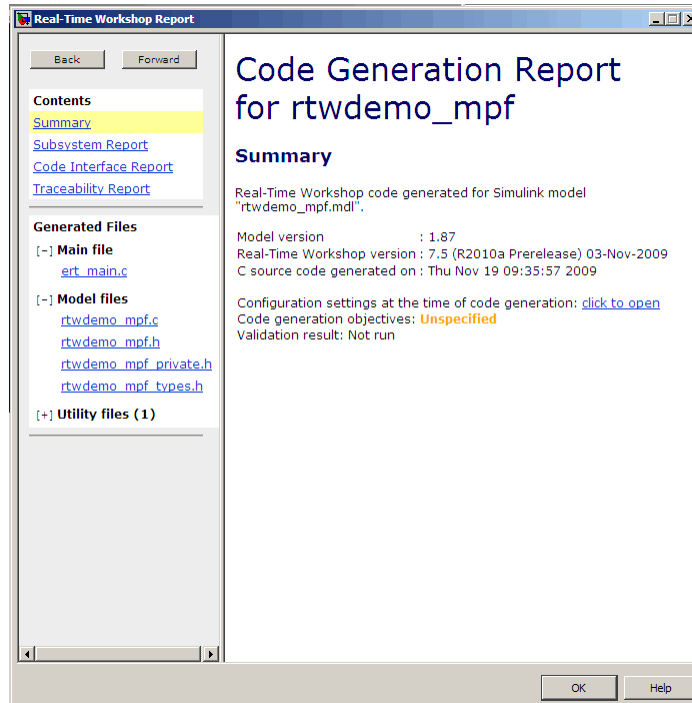
- 3 In the middle pane, select data objects one at a time, and notice their property values in the right pane. The figure below shows this for signal A. All of the data objects have default property values. Note that for an **mpt** data object, the default in the **Storage class** field is **Global (Custom)**. For descriptions of the properties on the Model Explorer, see “Setting Property Values for Simulink Data Objects” on page 11-8.



Generate and Inspect Code

- 1 In the left pane of the Model Explorer, expand the **rtwdemo_mpf** node.
- 2 In the left pane, click **Configuration (Active)**.
- 3 In the center pane, click **Real-Time Workshop**. The active Real-Time Workshop configuration parameters appear in the right pane.
- 4 Click the **Report** tab.
- 5 In the Report tab, select **Create code generation report**

- 6 Select the **General** tab. Select **Generate code only**, and then click **Generate code**. After a few moments, the names of the generated files are listed on the Real-Time Workshop Report on the left pane.



- 7 Open and inspect the content of the model source file `rtwdemo_mpf.c`. The following data objects in the data dictionary are initialized in this file.

```

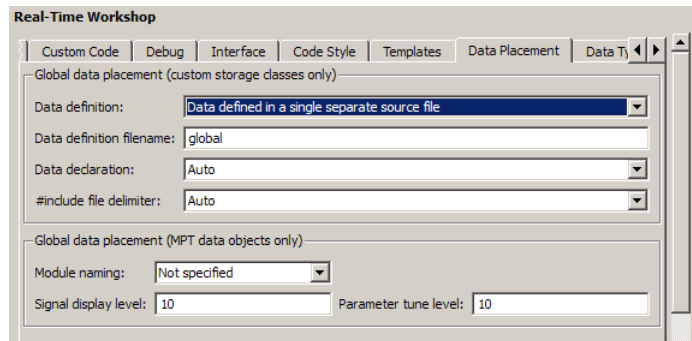
real_T F1 = 2.0;
real_T G1 = 6.0;
real_T G2 = -2.6;
real_T G3 = 9.0;
real_T Gain1 = 5.0;
real_T Gain2 = -3.0;

```

Defining All Global Data Objects in a Separate File

The previous procedure placed all of the model's data objects in the model source file. Now you place all of the global data objects in a file separate from the model source file:

- 1 Configure the model's generated code to include all Simulink data objects (signal and parameter) in a separate definition file. Set **Diagnostics > Data Validity > Signal resolution** to **Explicit** and **implicit**.
- 2 Specify that data be defined in a separate file. Set **Real-Time Workshop > Data Placement > Data definition** to **Data defined in single separate source file**. . Accept the default for **Data definition filename**, `global.c`



- 3 Specify that data be declared in a separate file. Set **Data declaration** to **Data declared in a single separate header file** and accept the default for **Data declaration filename**, `global.h`. Then, click **Apply**.
- 4 Click **Generate code**. Notice that the code generation report lists `global.c` and `global.h` files.
- 5 Inspect the code generation report. Notice that
 - The data objects formerly initialized in `rtwdemo_mpf.c` now are initialized in `global.c`.
 - The file `rtwdemo_mpf.c` includes `rtwdemo_mpf.h`.

- The file `rtwdemo_mpf.h` includes `global.h`.

Defining a Specific Global Data Object in Its Own File

The previous procedure placed all global data objects in a separate definition file, in one operation. You named that file `global.c`. (You named the corresponding declaration file `global.h`.) MPF allows you to override this and place a specific data object in its own definition file. In the following procedure, you move the `Final` signal to a file called `finalsig.c`, and keep all the other data objects defined in `global.c`:

- 1** In the Model Explorer, display the base workspace and select the `Final` signal object. The **mpt.Signal** properties appear in the right pane.
- 2** In the **Code generation options** section, type `finalsig.h` in the **Header file** text box, type `finalsig.c` in the **Definition file** text box, and click **Apply**.
- 3** On the Real-Time Workshop **General** pane, click **Generate code**. The code generation report still lists `global.c` and `global.h`, but adds `finalsig.c` and `finalsig.h`.
- 4** Open all four files to inspect them. Notice that the `Final` signal is defined in `finalsig.c`. All other data objects in the dictionary are defined in `global.c`.

Saving and Loading Data Objects

In a `.mat` file, you can save the set of data objects (and their properties) that you have created and load this information for later use or exchange it with another user. You can save some of the data objects in the workspace or all of them. See [Opening, Loading, Saving Files](#) in the MATLAB documentation.

Applying Naming Rules to Identifiers Globally

In this section...

“Overview” on page 11-30

“Changing Names of Identifiers” on page 11-31

“Specifying Simulink Data Object Naming Rules” on page 11-34

“Defining Rules That Change All Signal Names” on page 11-35

“Defining Rules That Change All Parameter Names” on page 11-35

“Defining Rules That Change All #defines” on page 11-36

Note The capabilities described in this section apply both to Simulink and mpt data objects.

Overview

Signal and parameter names appear on a Simulink model. The same names appear as data objects on the Model Explorer. By default, these names are replicated exactly in the generated code. For example, "Speed" on the model (and workspace) appears as the identifier "Speed" in the code, by default. But you can change how they appear in the code. For example, if desired, you can change "Speed" to SPEED or speed. Or, you can choose to use a different name altogether in the generated code, like MPH. The only restriction is that you follow ANSI C²/C++ rules for naming identifiers.

There are two ways of changing how a signal name or parameter name is represented in the generated code. You can do this *globally*, by following the procedure in this section. This procedure makes selections on the Configuration Parameters dialog box to change *all* of the names when code generation occurs, according to the same rule. Or, you can change the names *individually* by following the steps described in “Setting Property Values for Simulink Data Objects” on page 11-8. The relevant field in that procedure is **Alias** on the Model Explorer.

2. ANSI[®] is a registered trademark of the American National Standards Institute, Inc.

If the **Alias** field is empty, the naming rule that you select on the Configuration Parameters dialog box applies to all data objects. But if you do specify a name in the **Alias** field, this overrides the naming rule for that data object. The table below illustrates these cases. The table assumes that you selected Force lower case as the naming rule. But with the information provided, you can determine how any of the naming rules works for an mpt data object or a Simulink data object (Force upper case, Force lower case, or Custom M-function).

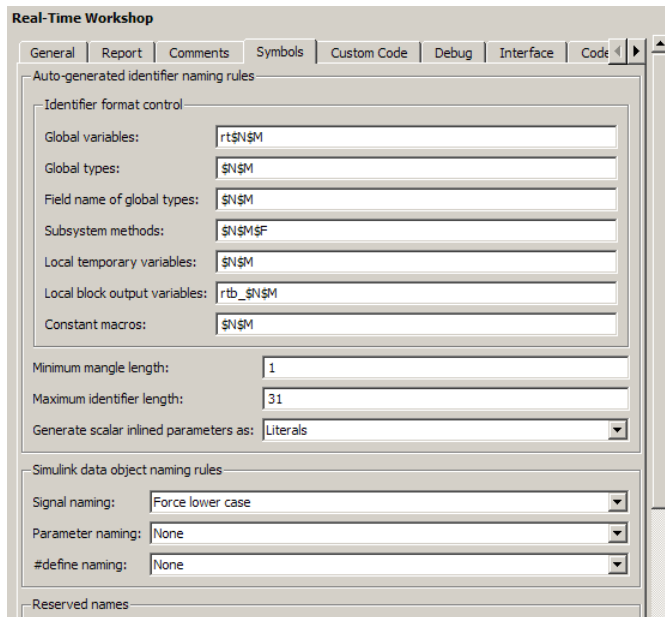
Naming Rules and Alias Override (Global Change of Force Lower Case Rule)

Name of Data Object in Model	Name in Alias Field	Package	Result in Generated Code
A		Simulink or mpt	a
A	D	Simulink or mpt	D

Changing Names of Identifiers

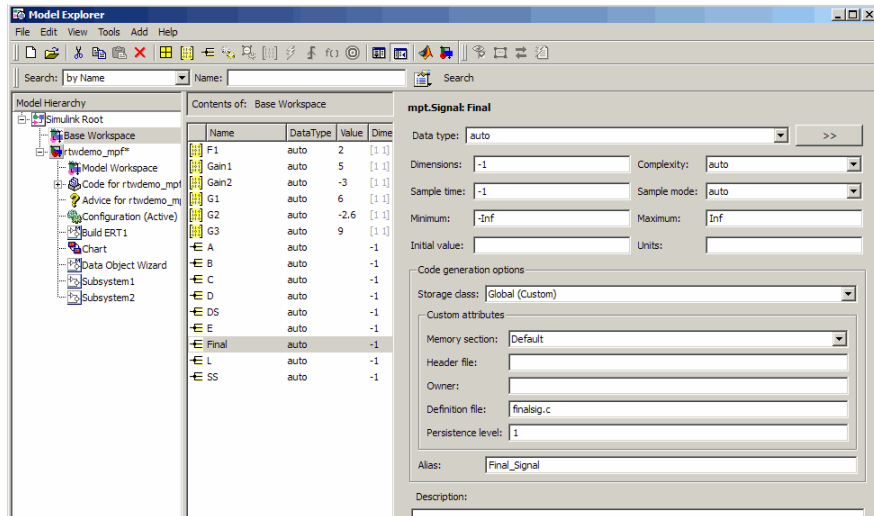
This procedure changes the names of all signal identifiers, except one, so that they are spelled with all lowercase letters. For example, A in the definition statement located in `global.c` is changed to `a`. The one exception is the `Final` signal in the `finalsig.c` file. You change this identifier name to `Final_Signal`. The names of the rest of the identifiers in the generated files remain the same:

- 1 Open the **Real-Time Workshop > Symbols** pane of the Configuration Parameters dialog.
- 2 In the **Simulink data object naming rules** section, set **Signal naming** to Force lower case, and click **Apply**.



3 Display the base workspace and select **Final**.

4 In the right pane, type **Final_Signal** in the **Alias** text box, then click **Apply**.



- 5 On the Real-Time Workshop **General** pane, click **Generate code** . Now, the signal identifiers in `global.c` and `global.h` appear with lowercase letters.

```

real_T F1 = 0.0;
real_T G1 = 1.0;
real_T G2 = 1.0;
real_T G3 = 1.0;
real_T Gain1 = 0.0;
real_T Gain2 = 0.0;
real_T a;
real_T b;
real_T c;
real_T d;
real_T ds;
real_T e;
real_T l;
real_T ss;

```

The statement defining the Final signal in `finalsig.c` looks like this:

```
real T Final_Signal;
```

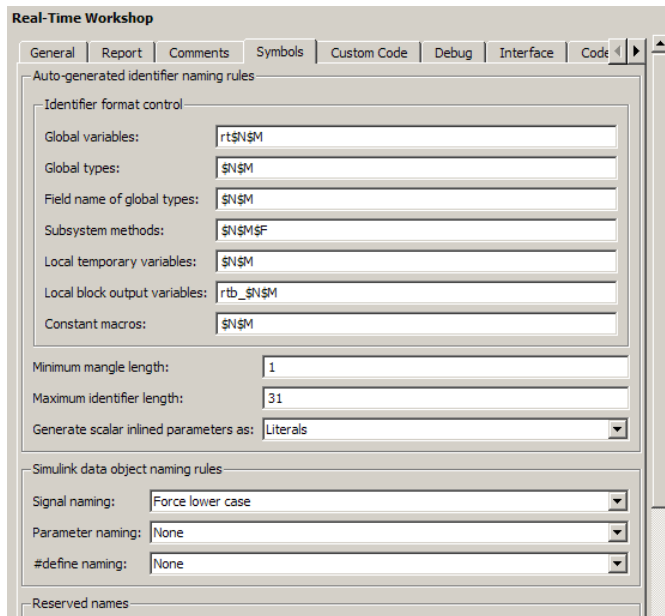
The statement declaring this identifier in `finalsig.h` looks like this:

```
extern real_T Final_Signal;
```

Specifying Simulink Data Object Naming Rules

You specify Simulink data object naming rules on the **Real-Time Workshop > Symbols** pane of the Configuration Parameters dialog box. To access that pane,

- 1 Open an ERT-based model.
- 2 Open the Configuration Parameters dialog box from the **Simulation** menu or Model Explorer.
- 3 Open the **Real-Time Workshop > Symbols** pane. See the subpane **Simulink data object naming rules**.



Notice the preconfigured settings on this pane. If all of these are acceptable as is, proceed to “Creating User Data Types” on page 11-38. Otherwise, follow the procedures below, as desired, to change signal names, parameter names, or parameter names that you want to use in a `#define` preprocessor

directive. “Real-Time Workshop Pane: Symbols” in the Real-Time Workshop documentation describes all fields on this pane and their possible settings for these procedures.

- “Defining Rules That Change All Signal Names” on page 11-35
- “Defining Rules That Change All Parameter Names” on page 11-35
- “Defining Rules That Change All #defines” on page 11-36

Defining Rules That Change All Signal Names

This procedure allows you to change all of the model’s signal names, using the same rule. The new names will appear as identifiers in the generated code:

- 1** In the **Signal naming** menu, click the desired selection. (“Signal naming” explains the possible selections.) The default is **None**. If you selected **Custom M-function**, go to the next step. Otherwise, click **Apply** and then generate and inspect code.
- 2** Write a MATLAB function that changes all occurrences of signal names in the model to appear the way you want as identifiers in the generated code. (An example is shown in “Defining Rules That Change All Parameter Names” on page 11-35.)
- 3** Save the function as a `.m` file in any folder that is in the MATLAB path.
- 4** In the **M-function** field under **Signal naming**, type the name of the file you saved in the previous step.
- 5** Click **Apply** and then generate and inspect code.

Defining Rules That Change All Parameter Names

This procedure allows you to change all of the model’s parameter names, using the same rule. The new names will appear as identifiers in the generated code:

- 1** In the **Parameter naming** field, click the desired selection. (“Parameter naming” explains the possible selections.) The default is **None**. If you selected **Custom M-function**, go to the next step. Otherwise, click **Apply**, and proceed to “Defining Rules That Change All Signal Names” on page 11-35.

- 2 Write a MATLAB function that changes all occurrences of parameter names in the model to appear the way you want as identifiers in the generated code. For example, the code below changes all parameter names as necessary to make their first letter uppercase, and their remaining letters lowercase.

```
function
revisedName = initial_caps_only(name, object)
% INITIAL_CAPS_ONLY: User-defined naming rule causing each
% identifier in the generated code to have initial cap(s).
%
% name: name as spelled in model.
% object: the object of name; includes name's properties.
%
% revisedName: manipulated name returned to MPT for the
code.
%
%
:
revisedName = [upper(name(1)),lower(name(2:end))];
:
```

- 3 Save the function as a `.m` file in any folder that is in the MATLAB path.
- 4 In the **M-function** field under **Parameter naming**, type the name of the file you saved in the previous step.
- 5 Click **Apply** and then define rules that apply to all signal names.

Defining Rules That Change All #defines

This procedure allows you to change all of the model's parameter names whose storage class you selected as Define in "Creating mpt Data Objects with Data Object Wizard" on page 11-11, using the same rule. The new names will appear as identifiers in the generated code:

- 1 In **#define naming**, click the desired selection. ("#define naming" explains the possible selections.) The default is None. If you select Custom M-function, go to the next step. Otherwise, click **Apply** and proceed to "Defining Rules That Change All Parameter Names" on page 11-35.

- 2** Write a MATLAB function that changes all occurrences of the parameter name whose storage class you specified as **Define** in “Creating mpt Data Objects with Data Object Wizard” on page 11-11 so that it appears the way you want as an identifier in the generated code. (An example is shown below.)
- 3** Save the function as a `.m` file in any folder that is in the MATLAB path.
- 4** In the **M-function** field under **#define naming**, type the name of the file you saved in the previous step.
- 5** Click **Apply** and then define rules that change all parameter names.

Creating User Data Types

In this section...

“Overview” on page 11-38

“Registering User Data Types Using `sl_customization.m`” on page 11-39

“Example User Data Type Customization Using `sl_customization.m`” on page 11-41

Note The capabilities described in this section apply both to Simulink and mpt data objects.

Overview

By default, MathWorks data types (such as `real32_T` and `uint8_T`) are used to define data in the generated code. If you prefer using your company-standard data types (such as `DBL` and `U8`), you can define user data types. To use this feature, you must register and create your data types so that the code generator can associate them with the corresponding MathWorks C/C++ data types. Then, the code generator will use your user data types in the generated code instead of the MathWorks C/C++ data types.

The Real-Time Workshop software automatically associates the MathWorks C/C++ data types with the equivalent ANSI³ C/C++ data types. If you want to use only the default MathWorks C/C++ data types, you do not need to register and create your own data types.

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

3. ANSI[®] is a registered trademark of the American National Standards Institute, Inc.

Once you have registered your user data types using `sl_customization.m`, you must create the `Simulink.AliasType` objects corresponding to your user data types. If your model references a user data type either directly (for example, in the output data type of a block) or indirectly (for example, a `Simulink.Signal` object data type is set to the user data type), you must create the corresponding `Simulink.AliasType` object before updating the model, running a simulation, or generating code. To create the `Simulink.AliasType` objects, you can:

- Invoke the MATLAB function `ec_create_type_obj` to programmatically create all the required `Simulink.AliasType` objects
- Create `Simulink.AliasType` objects one at a time by selecting **Add > Simulink.AliasType** in the Model Explorer
- Create `Simulink.AliasType` objects one at a time by entering the MATLAB command `userdatatype = Simulink.AliasType`, where `userdatatype` is a user data type registered in `sl_customization.m`

Registering User Data Types Using `sl_customization.m`

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

```
function sl_customization(cm)
```

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

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

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

- `addUserDataTypes(hObj, userName, builtinName, userHeader)`
`addUserDataTypes(hObj, userName, builtinName)`

```
addUserDataTypes(hObj, userName, aliasTypeObj)
addUserDataTypes(hObj, userName, numericTypeObj)
addUserDataTypes(hObj, userName, fixdtString)
```

Registers the specified user-defined data type and adds it to the top of the data type list, as displayed in the **Data type** pull-down list in the Model Explorer.

- `userName` — Name of the user data type
- `builtinName` — MathWorks C/C++ data type to which `userName` is mapped
- `userHeader` — Name of the user header file that includes the definition of the user data type
- `aliasTypeObj`, `numericTypeObj`, or `fixdtString` — `Simulink.AliasType`, `Simulink.NumericType`, or `fixdt` to which `userName` is mapped

Note If a `Simulink.AliasType` or `Simulink.NumericType` object of the same name as your registered user data type is already defined in the base workspace, the definitions of the existing object and the registered user data type must be consistent or a consistency warning will be displayed.

- `moveUserDataTypesToTop(hObj, userName)`

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

- `moveUserDataTypesToEnd(hObj, userName)`

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

- `removeUserDataTypes(hObj, userName)`

Removes the specified user-defined data type from the data type list.

Your instance of the `sl_customization` function should use these methods to register user data types for your Simulink installation.

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, you must restart your Simulink session or enter the following command at the MATLAB command line to effect the changes:

```
sl_refresh_customizations
```

Example User Data Type Customization Using `sl_customization.m`

The `sl_customization.m` file shown in Example 1: `sl_customization.m` for User Data Type Customizations on page 11-41 uses the following methods:

- `addUserDataTypes` to register the user-defined data types `MyInt16`, `MyInt32`, `MyBool`, and `fixdt(1,8)`
- `moveUserDataTypesToTop` to move `MyBool` to the top of the data type list, as displayed in the **Data type** pull-down list in the Model Explorer
- `removeUserDataTypes` to remove the built-in data types `boolean` and `double` from the data type list

Example 1: `sl_customization.m` for User Data Type Customizations

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

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

% Add custom types
hObj.addUserDataTypes('MyInt16', 'int16_T', '<mytypes.h>');
hObj.addUserDataTypes('MyInt32', 'int32_T', '<mytypes.h>');
hObj.addUserDataTypes('MyBool', 'boolean_T');
hObj.addUserDataTypes('fixdt(1,8)');

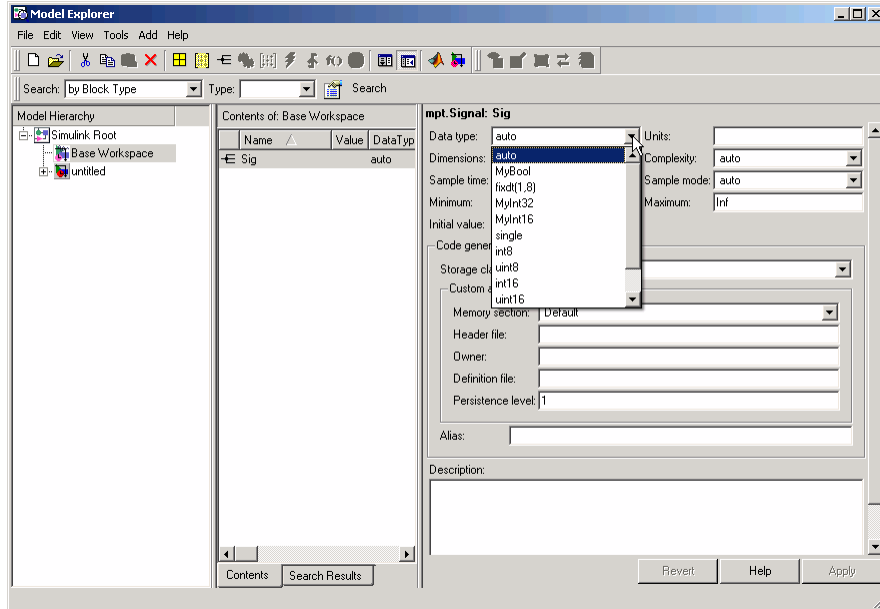
% Make MyBool first in the list
hObj.moveUserDataTypesToTop('MyBool');

% Remove built-in boolean and double from the list
hObj.removeUserDataTypes('boolean');
hObj.removeUserDataTypes('double');
```

end

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

- 1 Start a MATLAB session.
- 2 Open Model Explorer, for example, by entering the MATLAB command `daexplr`.
- 3 Select **Base Workspace**.
- 4 Add an `mpt` signal, for example, by selecting **Add > MPT Signal**.
- 5 In the right-hand pane display for the added `mpt` signal, examine the **Data type** drop-down list, noting the impact of the changes specified in Example 1: `sl_customization.m` for User Data Type Customizations on page 11-41.



Selecting User Data Types for Signals and Parameters

In this section...

“Preparing User Data Types” on page 11-43

“Selecting the User Data Types” on page 11-45

Preparing User Data Types

You can use user-defined data types for Simulink signals and parameters and their corresponding identifiers in generated code. This is true whether or not a signal or parameter has a Simulink data object associated with it.

Before you can select a user data type for a signal or parameter, you must:

- 1 Create a user data type (alias), as explained in the description of `Simulink.AliasType` in the Simulink documentation. For the example in “Selecting the User Data Types” on page 11-45 demonstrating how to select user data types for signals and parameters, create the alias data type `f32`.
- 2 Register the user data type so that it is associated with the corresponding MathWorks C/C++ data type, as explained in “Creating User Data Types” on page 11-38. For the example, register the data type `f32` so that it is associated with type `real132_T`. The call to function `addUserDataTypes` in the `s1_customization.m` file you use for the registration must specify:
 - `f32` as the user data type
 - `real132_T` as the built-in data type
 - `<userdata_types.h>` as the user header file that is to include the user data type definition

For example,

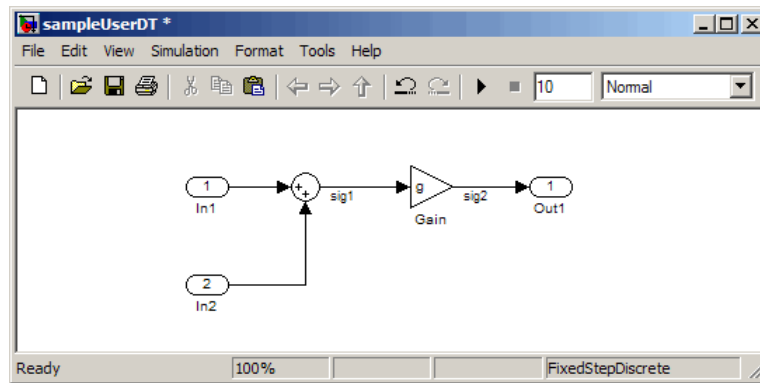
```
function s1_customization(cm)

hObj = cm.slDataObjectCustomizer;

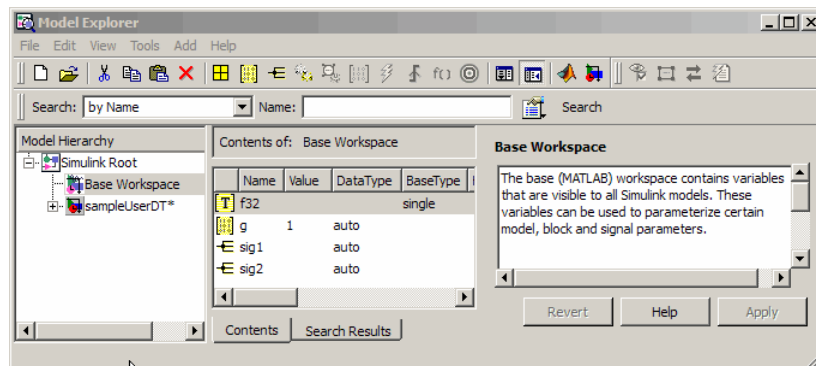
addUserDataTypes(hObj, 'f32', 'real132_T', '<userdata_types.h>');
```

end

- 3 If you have not already done so, add the directory containing the `sl_customization.m` file that you created or modified in step 1 to the MATLAB search path.
- 4 Open a model. The example uses the following model.



- 5 Create a data dictionary for the model, as explained in “Creating Simulink and mpt Data Objects” on page 11-4, to associate signals and parameters with data objects. For the example, the Model Explorer display must appear as shown below. The three data objects that appear, `sig1`, `sig2`, and `g`, and the registered user data type, `f32`, appear in the middle pane. The "T" indicates that `f32` is an alias data type.



For the selection procedure and to continue with the example, continue to “Selecting the User Data Types” on page 11-45.

Selecting the User Data Types

After completing the preparation explained in “Preparing User Data Types” on page 11-43, you can use the user-defined data types for Simulink signals and parameters and for their corresponding identifiers in the generated code. You can use user-defined data types with signals and parameters whether or not they have Simulink objects associated with them.

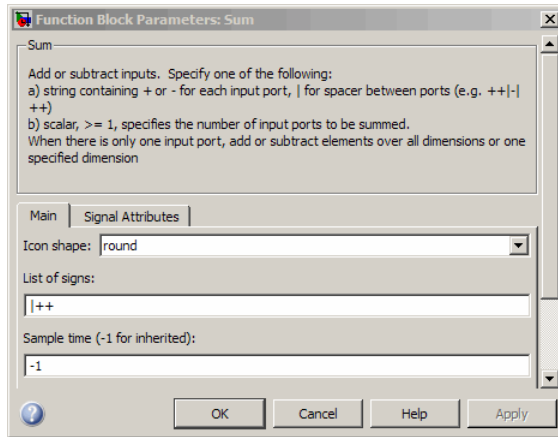
- 1 For an `mpt` object that is associated with a signal or parameter in your model, in the **Data type** field, select the user data type that you want. For example, select `f32`, for `sig1`.

This selects `f32` for the `sig1` data object in the data dictionary, but does not select `f32` for the corresponding labeled signal in the model. Therefore, the two may be in conflict. If you try to update the model, you could get an error message.

To continue with the example, type `f32` into the **Data type** field for `sig1`.

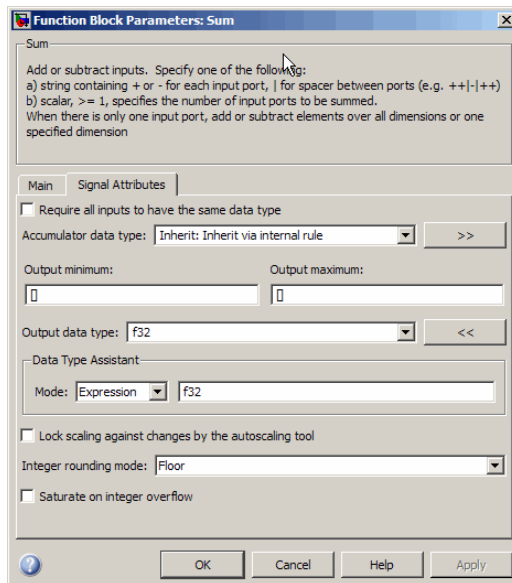
- 2 Select the model and double-click the signal or parameter source block. (The source block of a model signal or parameter controls the signal’s or parameter’s data type.) For example, in the example model the Sum block is the source block for `sig1`. Double-click the Sum block.

The Function Block Parameters dialog box opens.

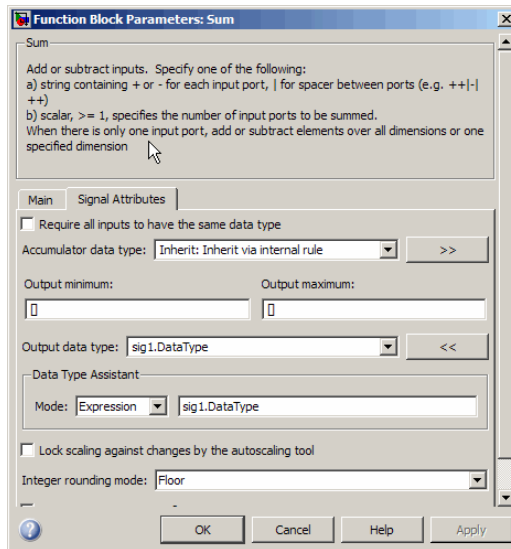


3 Select the **Signal Attributes** tab.

4 In the **Output data type** or **Parameter data type** field, type the name of the user data type that you specified for the data object in step 1, and click **Apply**. The user data type of the signal in the model and that of the signal object are now the same.



Alternatively, you can use dictionary-driven data typing. In the **Output data type** field, specify `object.DataType`, where `object` is the case-sensitive object name. For example, you can specify `sig1.DataType` instead of `f32`.



The advantage of using the alternative is that subsequent user data type changes for the object in the dictionary automatically change the user data type of the corresponding model signal or parameter.

- 5 Repeat steps 1 through 4 for each remaining model signal and parameter that has a corresponding signal object for which you selected a user data type.
- 6 Save the model and save all of the data objects in the MATLAB base workspace in a `.mat` file for reuse later. Generated code for `sig1` in the example model (with default MPF settings) would appear as follows:

```
In sampleUserDT.c           f32 sig1;
In sampleUserDT_types.h    #include <userdata_types.h>
```

Registering mpt User Object Types

In this section...

“Introduction” on page 11-48

“Registering mpt User Object Types Using `sl_customization.m`” on page 11-48

“Example mpt User Object Type Customization Using `sl_customization.m`” on page 11-50

Introduction

Real-Time Workshop Embedded Coder software allows you to create custom mpt object types and specify properties and property values to be associated with them (see “Creating mpt Data Objects with Data Object Wizard” on page 11-11). Once created, a user object type can be applied to data objects displayed in Model Explorer. When you apply a user object type to a data object, by selecting a type name in the **User object type** pull-down list in Model Explorer, the data object is automatically populated with the properties and property values that you specified for the user object type.

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

Registering mpt User Object Types Using `sl_customization.m`

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


```
function sl_customization(cm)
```

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

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

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

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

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

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

- `objectTypeName` — Name of the user object type
- `classType` — Class to which the user object type applies: 'Signal', 'Parameter', or 'Both'
- `propName` — Name of a property of an mpt or mpt-derived data object to be populated with a corresponding `propValue` when the registered user object type is selected
- `propValue` — Specifies the value for a corresponding `propName`
- `moveMPTObjectTypeToTop(hObj, objectTypeName)`
 Moves the specified user object type to the top of the user object type list, as displayed in the **User object type** pull-down list in the Model Explorer.
- `moveMPTObjectTypeToEnd(hObj, objectTypeName)`
 Moves the specified user object type to the end of the user object type list, as displayed in the **User object type** pull-down list in the Model Explorer.
- `removeMPTObjectType(hObj, objectTypeName)`

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

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

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

Example mpt User Object Type Customization Using `sl_customization.m`

The `sl_customization.m` file shown in Example 2: `sl_customization.m` for mpt Object Type Customizations on page 11-50 uses the `addMPTObjectType` method to register the user signal types `EngineType` and `FuelType` for mpt objects.

Example 2: `sl_customization.m` for mpt Object Type Customizations

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

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

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

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

end

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

- 1** Start a MATLAB session.
- 2** Open Model Explorer, for example, by entering the MATLAB command `daexplr`.
- 3** Select **Base Workspace**.
- 4** Add an `mpt` signal, for example, by selecting **Add > MPT Signal**.
- 5** In the right-hand pane display for the added `mpt` signal, examine the **User object type** drop-down list, noting the impact of the changes specified in Example 2: `sl_customization.m` for `mpt` Object Type Customizations on page 11-50.
- 6** From the **User object type** drop-down list, select one of the registered user signal types, for example, `FuelType`, and verify that the displayed settings are consistent with the arguments specified to the `addMPTObjectType` method in `sl_customization.m`.

Replacing Built-In Data Type Names in Generated Code

In this section...

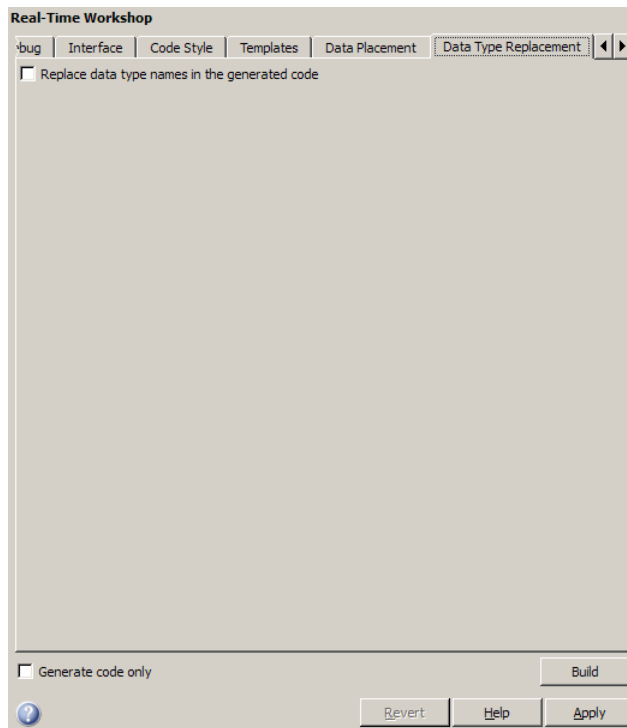
“Replacing Built-In Data Type Names” on page 11-52

“Replacing boolean with an Integer Data Type” on page 11-58

“Data Type Replacement Limitations” on page 11-60

Replacing Built-In Data Type Names

If your application requires you to replace built-in data type names with user-defined replacement data type names in the generated code, you can do so from the **Real-Time Workshop > Data Type Replacement** pane of the Configuration Parameters dialog box, shown below in the Model Explorer view.

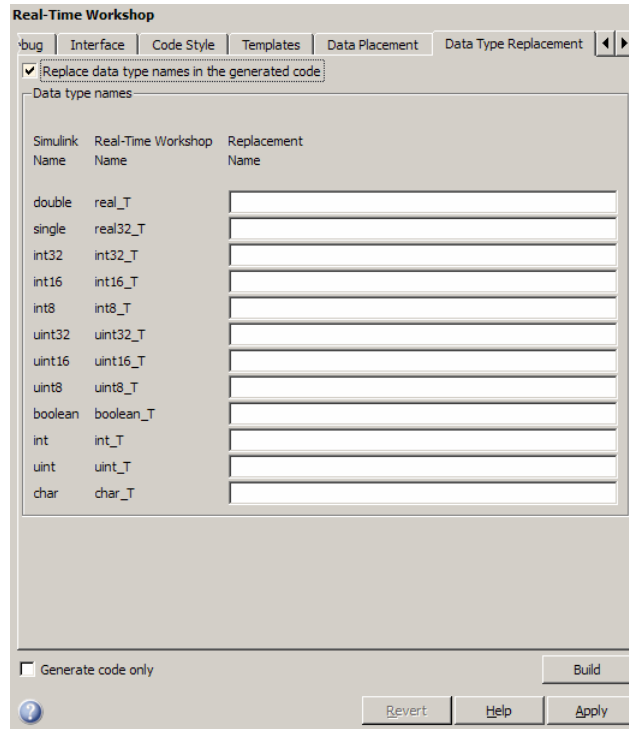


This pane is available for ERT target based Simulink models. In addition to providing a mechanism for mapping built-in data types to user-defined replacement data types, this feature:

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

Note For limitations that apply, see “Data Type Replacement Limitations” on page 11-60.

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



The table **Data type names** lists each Simulink built-in data type name along with its Real-Time Workshop data type name. Selectively fill in fields in the third column with your replacement data types. Each replacement data type should be the name of a `Simulink.AliasType` object that exists in the base workspace. Replacements may be specified or not for each individual built-in type. For each replacement data type entered, the `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.

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

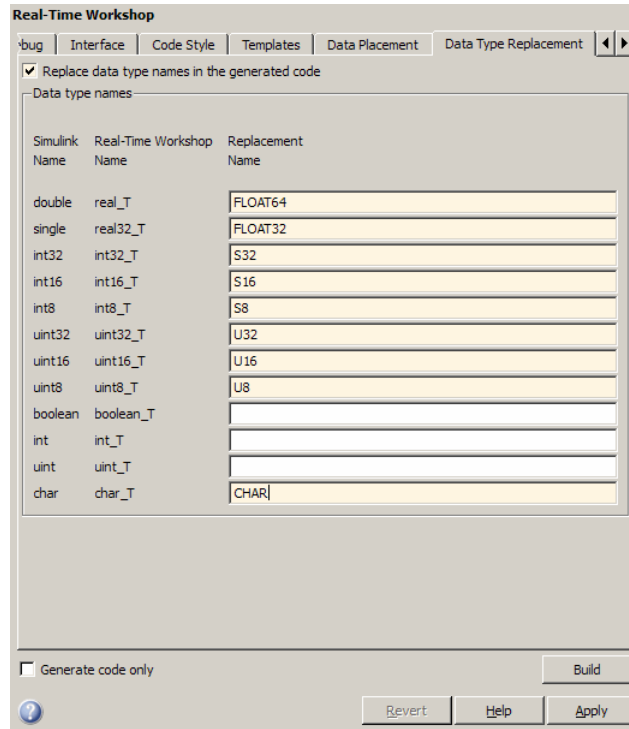
- For `int`, `uint`, and `char`, the replacement data type's size must match the size displayed for **Number of bits: int** or **Number of bits: char** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

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

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

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

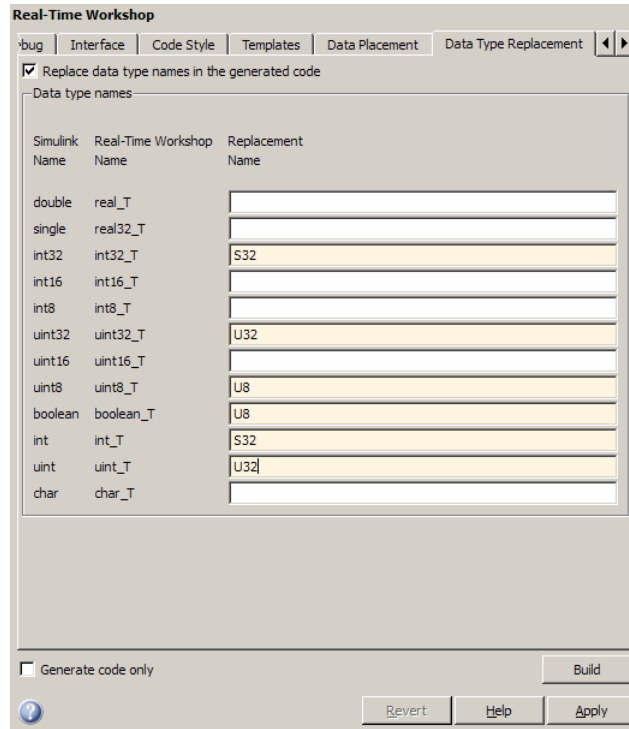
You can fill in the **Data Type Replacement** pane with a one-to-one replacement mapping, as follows:



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

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

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



The user-defined replacement types you specify will appear in your model's generated code in place of the corresponding built-in data types. For example, if you specify user-defined data type `FLOAT64` to replace built-in data type `real_T` (double), then the original generated code shown in Example 3: Generated Code with `real_T` Built-In Data Type on page 11-58 will become the modified generated code shown in Example 4: Generated Code with `FLOAT64` Replacement Data Type on page 11-58.

Example 3: Generated Code with real_T Built-In Data Type

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

Example 4: Generated Code with FLOAT64 Replacement Data Type

```
...
/* Model initialize function */
void sinwave_initialize(void)
{
...
  {FLOAT64 *dwork_ptr = (FLOAT64 *) &sinwave_DWork.lastSin;
...
}
...
```

Replacing boolean with an Integer Data Type

Using data type replacement, you can replace the `boolean` built-in data type in generated code with any integer type, among the following, that might improve the performance of generated code on your production hardware:

- `int8`
- `uint8`
- `int n`
- `uint n`

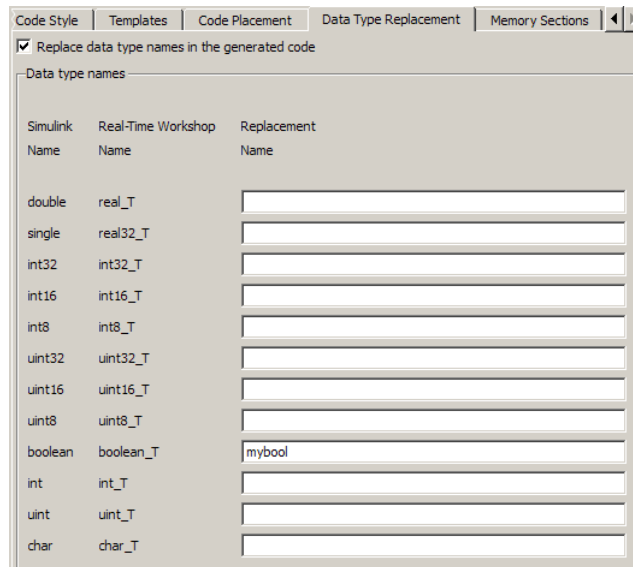
where n is 8, 16, or 32, matching the integer word size for the production hardware (for example, `int32` for 32-bit hardware).

For example, to map `boolean` to the `int32` data type, perform the following steps.

- 1 Define a `Simulink.AliasType` object with a base type of `int32`:

```
mybool = Simulink.AliasType;
mybool.BaseType = 'int32';
```

- 2 Open an ERT-based model. In the **Data Type Replacement** pane of the Configuration Parameters dialog box, map `boolean` (`boolean_T`) to the replacement data type `mybool`.



In the resulting generated code, `boolean` is replaced with `mybool`. For example, `rtwtypes.h` contains:

```
/* Generic type definitions ... */
...
typedef int boolean_T;
...
/* Define RTW replacement data types. */
typedef boolean_T mybool; /* User defined replacement datatype for boolean_T */
```

Boolean data in the generated code is declared with `mybool`. For example, given a model with a Boolean output `Out1`, an `Out1` declaration such as

```
boolean_T Out1;                                /* '<Root>/Out1' */
```

instead is generated in *model.h* as

```
mybool Out1;                                   /* '<Root>/Out1' */
```

Data Type Replacement Limitations

- Data type replacement does not support multiple levels of mapping. Each replacement data type name maps directly to one or more built-in data types.
- Data type replacement is not supported for simulation target code generation for referenced models.
- Data type replacement is not supported if the **GRT compatible call interface** option is selected for your model.
- Data type replacement occurs during code generation for all *.c*, *.cpp*, and *.h* files generated in build directories (including top and referenced model build directories) and in the *_sharedutils* directory. *Exceptions* are as follows:

```
rtwtypes.h  
model_sf.c or .cpp (ERT S-function wrapper)  
model_dt.h (C header file supporting external mode)  
model_capi.c or .cpp  
model_capi.h
```

- Data type replacement is not supported for complex data types.
- Many-to-one data type replacement is not supported for the *char* built-in data type. Attempting to use *char* as part of a many-to-one mapping to a user-defined data type introduces a violation of the MISRA C[®] specification. Specifically, if *char* (*char_T*) and either *int8* (*int8_T*) or *uint8* (*uint8_T*) are mapped to the same user replacement type, the result is a MISRA C violation. Additionally, if you try to generate C++ code, invalid implicit type casts are made and compile-time errors may result. Use *char* only in one-to-one data type replacements.

Customizing Data Object Wizard User Packages

In this section...

“Introduction” on page 11-61

“Registering Data Object Wizard User Packages Using `sl_customization.m`” on page 11-61

“Example Data Object Wizard User Package Customization Using `sl_customization.m`” on page 11-63

Introduction

Data Object Wizard (DOW) can be run in connection with a Simulink model to quickly determine which model data are not associated with data objects and to create and associate data objects with the data. (For more information about Data Object Wizard, see “Data Object Wizard” in the Simulink documentation and “Creating Simulink Data Objects with Data Object Wizard” on page 11-5.) If you want the wizard to use data object classes from a package other than the standard Simulink class package to create the data objects, you select the package from the wizard’s **Choose package for selected data objects** list. This package list can be customized in various ways, including adding or removing packages and modifying the list order.

To register Data Object Wizard user package customizations, use the Simulink customization file `sl_customization.m`. This file is a mechanism that allows you to use MATLAB code to perform customizations of the standard Simulink user interface. The Simulink software reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see “Customizing the Simulink User Interface” in the Simulink documentation.

Registering Data Object Wizard User Packages Using `sl_customization.m`

To register Data Object Wizard user package customizations, you create an instance of `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization`

function accepts one argument: a handle to a customization manager object. For example,

```
function sl_customization(cm)
```

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

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

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

- `addUserPackage(hObj, packageName)`
`addUserPackage(hObj, cellArrayOfStrings)`
Adds the specified user package(s) to the top of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard.
- `moveUserPackageToTop(hObj, packageName)`
Moves the specified user package to the top of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard.
- `moveUserPackageToEnd(hObj, packageName)`
Moves the specified user package to the end of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard.
- `removeUserPackage(hObj, packageName)`
Removes the specified user package from the package list.
- `setUserPackages(hObj, cellArrayOfStrings)`
Replaces the entire package list with a specified list of user packages.

Your instance of the `sl_customization` function should use these methods to register DOW user package customizations for your Simulink installation.

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, you must restart your Simulink session or enter the following command at the MATLAB command line to effect the changes:

```
sl_refresh_customizations
```

Example Data Object Wizard User Package Customization Using `sl_customization.m`

The `sl_customization.m` file shown in Example 5: `sl_customization.m` for DOW User Package Customizations on page 11-63 uses the following methods:

- `addUserPackage` to add the user packages `ECoderDemos` and `SimulinkDemos` (present by default in the MATLAB path) to the top of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard
- `moveUserPackageToEnd` to move `SimulinkDemos` to the end of the package list

Example 5: `sl_customization.m` for DOW User Package Customizations

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

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

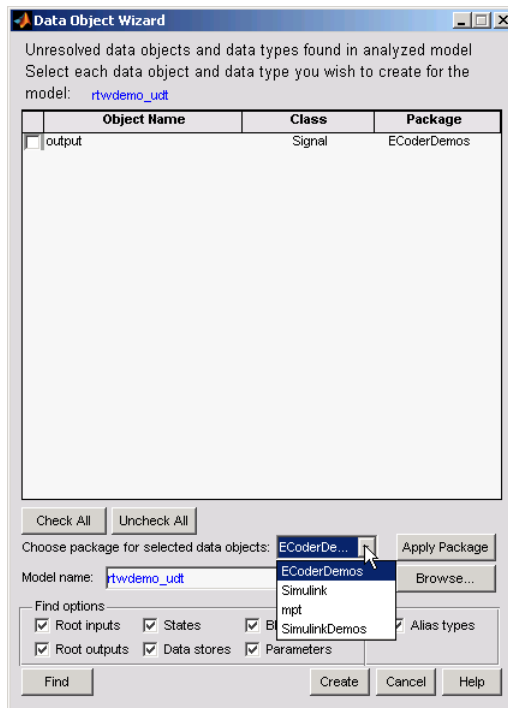
% Add user packages
hObj.addUserPackage({'ECoderDemos', 'SimulinkDemos'});

% Move SimulinkDemos to end of list
hObj.moveUserPackageToEnd('SimulinkDemos');

end
```

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

- 1 Start a MATLAB session.
- 2 Launch a model, such as `rtwdemo_udt`.
- 3 Open Data Object Wizard, for example, by selecting **Tools > Data Object Wizard** in the Simulink window.
- 4 In the Data Object Wizard dialog box, click the **Find** button to generate a list of one or more data objects.
- 5 Examine the **Choose package for selected data objects** drop-down list, noting the impact of the changes specified in Example 5: `sl_customization.m` for DOW User Package Customizations on page 11-63.



To replace the entire Data Object Wizard package list with a specified list of user packages, you can use a method invocation similar to the following:

```
hObj.setUserPackages({'myPackage1', 'ECoderDemos', 'mpt'});
```


Managing Placement of Data Definitions and Declarations

- “Overview of Data Placement” on page 12-2
- “Priority and Usage” on page 12-3
- “Ownership Settings” on page 12-10
- “Memory Section Settings” on page 12-11
- “Data Placement Rules” on page 12-12
- “Example Settings” on page 12-13
- “Data Placement Rules and Effects” on page 12-22

Overview of Data Placement

This chapter focuses on module packaging features (MPF) settings that are interdependent. Their combined values, along with Simulink partitioning, determine the file placement of data definitions and declarations, or *data placement*. This includes

- The number of files generated.
- Whether or not the generated files contain definitions for a model's global identifiers. And, if a definition exists, the settings determine the files in which MPF places them.
- Where MPF places global data declarations (*extern*).

The following six MPF settings are distributed among the main procedures and form an important interdependency:

- The **Data definition** field on the **Data Placement** pane of the Configuration Parameters dialog box.
- The **Data declaration** field on the **Data Placement** pane of the Configuration Parameters dialog box.
- The **Owner** field of the data object in the Model Explorer, and the **Module naming** and **Module name** fields on the **Data Placement** pane of the Configuration Parameters dialog box. The term "ownership settings" refers to **Owner**, **Module naming**, and **Module name** together.
- The **Definition file** field of the data object on the Model Explorer.
- The **Header file** field of the data object on the Model Explorer.
- The **Memory section** field of the data object on the Model Explorer.

Priority and Usage

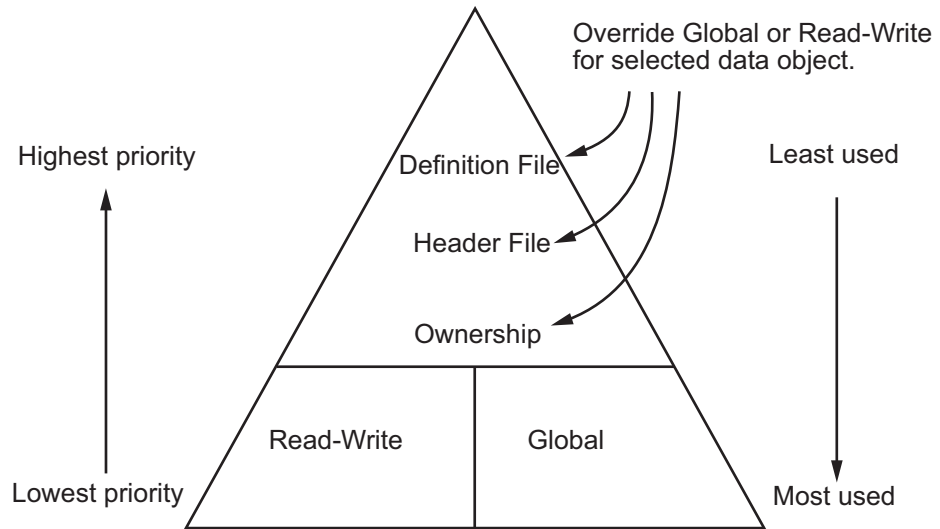
In this section...
“Overview” on page 12-3
“Read-Write Priority” on page 12-5
“Global Priority” on page 12-7
“Definition File, Header File, and Ownership Priorities” on page 12-9

Overview

There is a priority order among interdependent MPF settings. From highest to lowest, the priorities are

- Definition File priority
- Header File priority
- Ownership priority
- Read-Write priority or Global priority

Priority order varies inversely with frequency of use, as illustrated below. For example, Definition File is highest priority but least used.



MPF Settings Priority and Usage

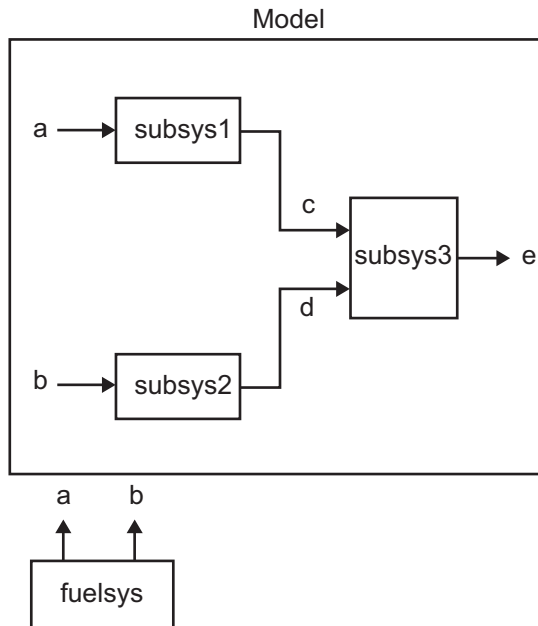
Unless they are overridden, the Read-Write and Global priorities place in the generated files *all* of the model's MPF-derived data objects that you selected using Data Object Wizard. (See “Creating Simulink Data Objects with Data Object Wizard” on page 11-5 for details.) Before generating the files, you can use the higher priority Definition file, Header file, and Ownership, as desired, to override Read-Write or Global priorities for single data objects. Most users will employ Read-Write or Global, without an override. A few users, however, will want to do an override for certain data objects. We expect that those users whose applications include multiple modules will want to use the Ownership priority.

The priorities are in effect only for those data objects that are derived from `Simulink.Signal` and `Simulink.Parameter`, and whose custom storage classes are specified using the Custom Storage Class Designer. (For details, see “Designing Custom Storage Classes and Memory Sections” on page 7-12 in the Real-Time Workshop Embedded Coder documentation.) Otherwise, the Real-Time Workshop build process determines the data placement.

Read-Write Priority

This is the lowest priority. Consider that a model consists of one or more Simulink blocks or Stateflow diagrams. There can be subsystems within these. For the purpose of illustration, think of a model with one top-level block called `fuelsys`. You double-clicked the block and now see three subsystems labeled `subsys1`, `subsys2` and `subsys3`, as shown in the next figure. Signals `a` and `b` are outputs from the top-level block (`fuelsys`). Signal `a` is an input to `subsys1` and `b` is input to `subsys2`. Signal `c` is an output from `subsys1`. Notice the other inputs and outputs (`d` and `e`). Signals `a` through `e` have corresponding data objects and are part of the code generation data dictionary.

As explained in Chapter 11, “Managing Data Definitions and Declarations With the Data Dictionary”, MPF provides you with the means of selecting a data object that you want defined as an identifier in the generated code. MPF also allows you to specify property values for each data object. For this illustration, we choose to include all of the data objects to be in the dictionary.

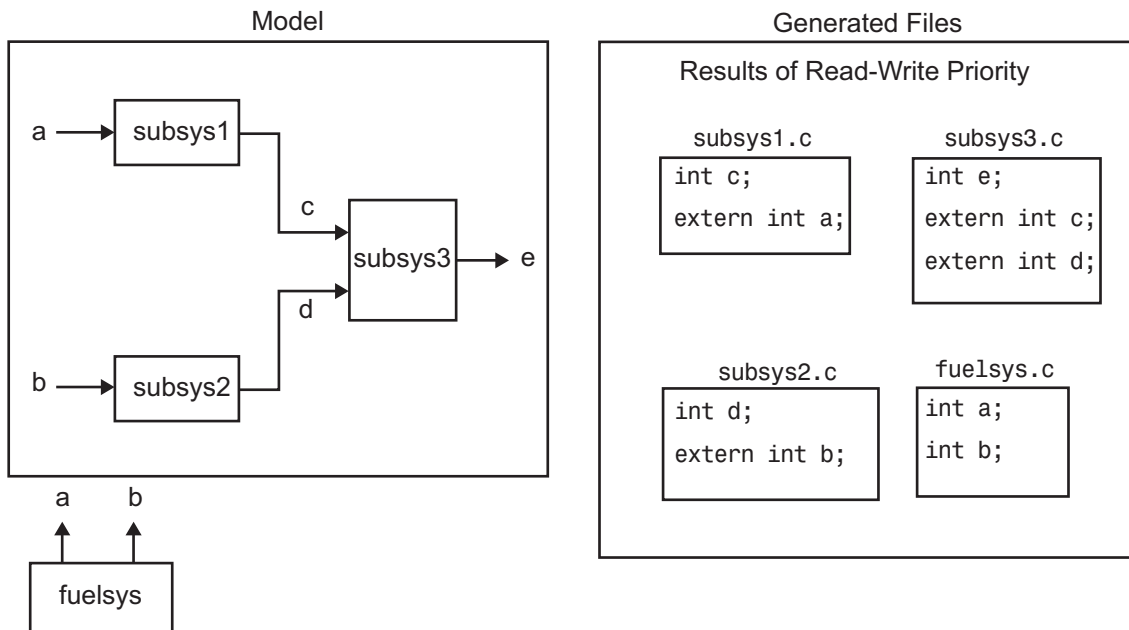


The Generated Files

We generate code for this model. As shown in the figure below, this results in a .c source file corresponding to each of the subsystems. (In actual applications, there could be more than one .c source file for a subsystem. This is based on the file partitioning previously selected for the model. But for our illustration, we only need to show one for each subsystem.) Data objects a through e have corresponding identifiers in the generated files.

A .c source file has one or more functions in it, depending on the internal operations (functions) of its corresponding subsystem. An identifier in a generated .c file has local scope when it is used only in one function of that .c file. An identifier has file scope when more than one function in the same .c file uses it. An identifier has global scope when more than one of the generated files uses it.

A subsystem's source file always contains the definitions for all of that subsystem's data objects that have local scope or file scope. (These definitions are not shown in the figure.) But where are the definitions and declarations for data objects of global scope? These are shown in the next figure.



When the Read-Write priority is in effect, this source file contains the definitions for the subsystem's global data objects, if this is the file that first writes to the data object's address. Other files that read (use) that data object only include a reference to it. This is why this priority is called Read-Write. Since a read and a write of a file are analogous to input and output of a model's block, respectively, there is another way of saying this. The definitions of a block's global data objects are located in the corresponding generated file, if that data object is an output from that block. The declarations (`extern`) of a block's global data objects are located in the corresponding generated file, if that data object is an input to that block.

Settings for Read-Write Priority

The generated files and what they include, as just described, occur when the Read-Write priority is in effect. For this to be the case, the other priorities are turned off. That is,

- The **Data definition** field on the **Data Placement** pane is set to `Data defined in source file`.
- The **Data declaration** field on the **Data Placement** pane is set to `Data declared in source file`.
- The **Owner** field on the Model Explorer is blank, and the **Module naming** field on the **Data Placement** pane is set to `Not specified`. (When `Not specified` is selected, the **Module name** field does not appear.)
- **Definition file** and **Header file** on the Model Explorer are blank.

Global Priority

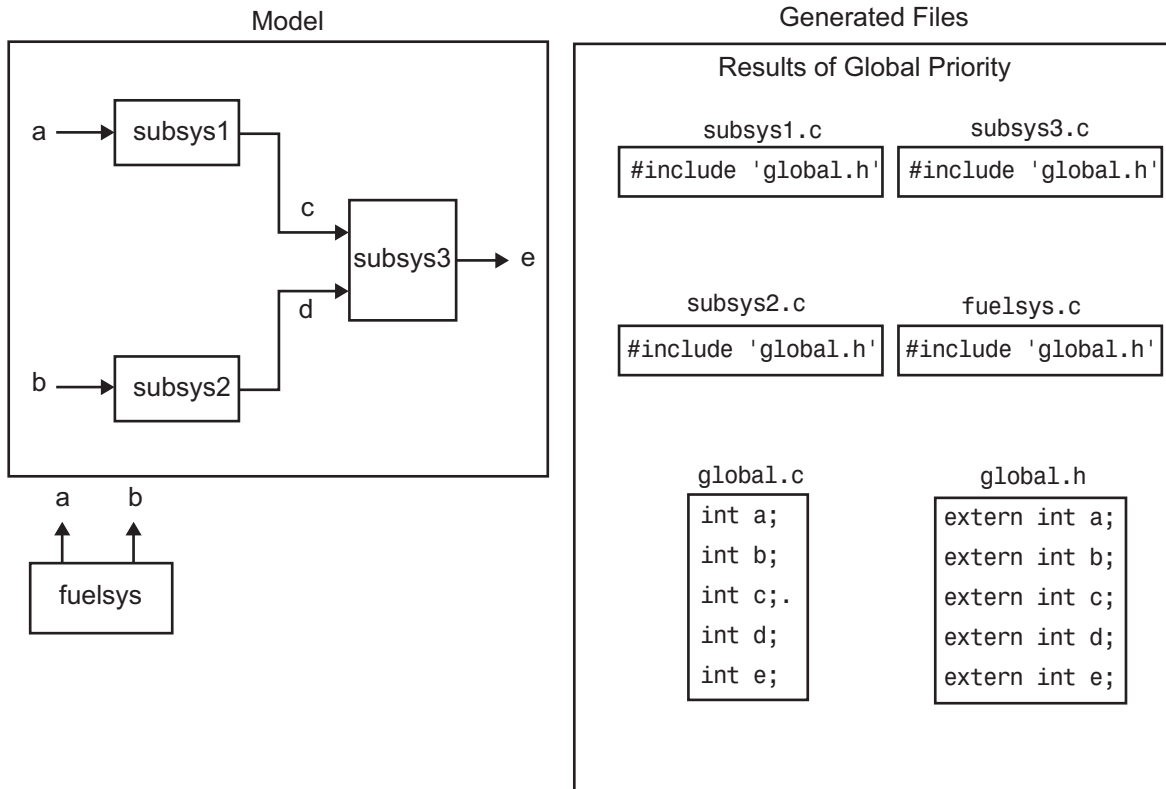
This has the same priority as Read-Write (the lowest) priority. The settings for this are the same as for Read-Write Priority, except

- The **Data definition** field on the **Data Placement** pane is set to `Data defined in single separate source file`.
- The **Data declaration** field on the **Data Placement** pane is set to `Data declared in single separate header file`.

The generated files that result are shown in the next figure. A subsystem's data objects of local or file scope are defined in the `.c` source file where the subsystem's functions are located (not shown). The data objects of global

scope are defined in another .c file (called `global.c` in the figure). The declarations for the subsystem's data objects of global scope are placed in a .h file (called `global.h`).

For example, all data objects of local and file scope for `subsys1` are defined in `subsys1.c`. Signal `c` in the model is an output of `subsys1` and an input to `subsys2`. So `c` is used by more than one subsystem and thus is a global data object. Since global priority is in effect, the definition for `c` (`int c;`) is in `global.c`. The declaration for `c` (`extern int c;`) is in `global.h`. Since `subsys2` uses (reads) `c`, `#include "global.h"` is in `subsys2.c`.



Definition File, Header File, and Ownership Priorities

While the Read-Write and Global priorities operate on *all* MPF-derived data objects that you want defined in the generated code, the remaining priorities allow you to override the Read-Write or Global priorities for one or more particular data objects. There is a high-to-low priority among these remaining priorities — Definition File, Header File, and Ownership — for a particular data object, as shown in MPF Settings Priority and Usage on page 12-4

Ownership Settings

Ownership settings refers to the values specified for the **Module naming** and **Module names** fields on the **Data Placement** pane of the Configuration Parameters dialog box, and the **Owner** field of a data object in the Model Explorer. These settings have no effect on what files are generated. Their effects only have to do with definitions and `extern` statements. There are five possible configurations, as shown in “Effects of Ownership Settings” on page 12-22.

Memory Section Settings

Memory sections allow you to specify storage directives for a data object. As shown in Parameter and Signal Property Values on page 6-2, the possible values for the **Memory section** property of a parameter or signal object are Default, MemConst, MemVolatile or MemConstVolatile.

If you specify a filename for **Definition file**, and select Default, MemConst, MemVolatile or MemConstVolatile for the **Memory section** property, the Real-Time Workshop Embedded Coder software generates a .c file and an .h file. The .c file contains the definition for the data object with the pragma statement or qualifier associated with the **Memory section** selection. The .h file contains the declaration for the data object. The .h file can be included, using the preprocessor directive `#include`, in any file that needs to reference the data object.

You can add more memory sections. For more information, see “Designing Custom Storage Classes and Memory Sections” on page 7-12 and Chapter 8, “Inserting Comments and Pragmas in Generated Code”.

Data Placement Rules

For a complete set of data placement rules in convenient tabular form, based on the priorities discussed in this chapter, see “Data Placement Rules and Effects” on page 12-22.

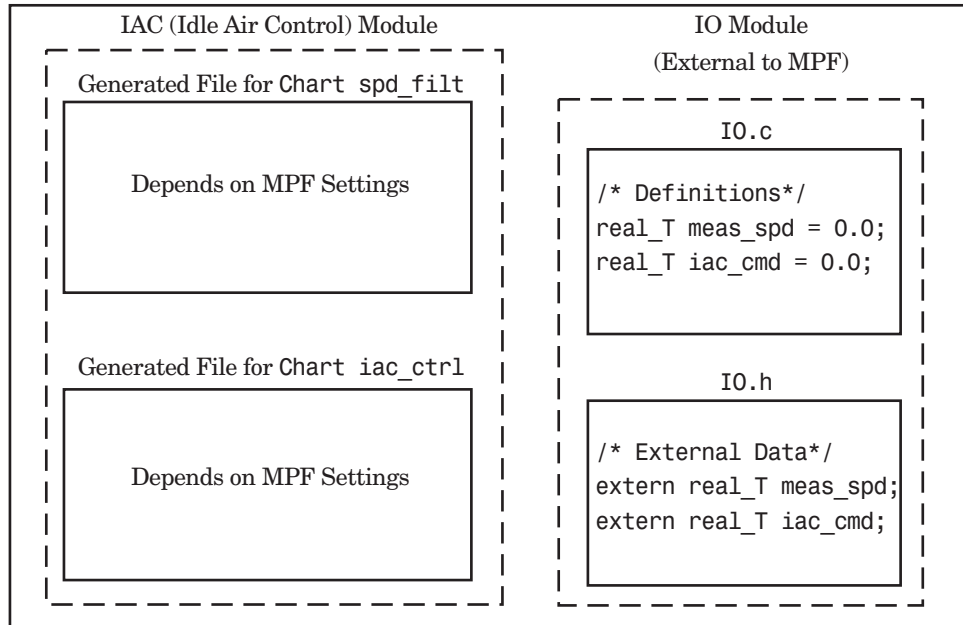
Example Settings

In this section...
“Introduction” on page 12-13
“Read-Write Example” on page 12-15
“Ownership Example” on page 12-17
“Header File Example” on page 12-18
“Definition File Example” on page 12-20

Introduction

“Example Settings and Resulting Generated Files” on page 12-23 provides example settings for one data object of a model. Eight examples are listed so that you can see the generated files that result from a wide variety of settings. Four examples from this table are discussed below in more detail. These discussions provide adequate information for understanding the effects of any settings you might choose. For illustration purposes, the four examples assume that we are dealing with an overall system that controls engine idle speed.

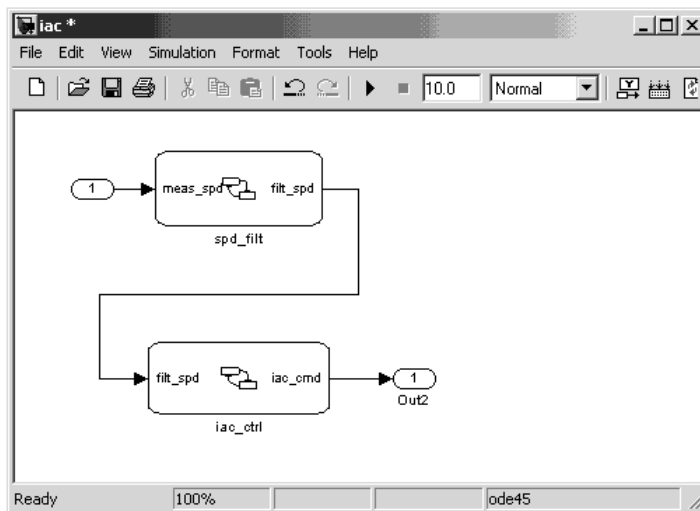
The next figure shows that the software component of this example system consists of two modules, IAC (Idle Air Control), and IO (Input-Output).



Engine Idle Speed Control System

The code in the IO module controls the system's IO hardware. Code is generated only for the IAC module. (Some other means produced the code for the IO module, such as hand-coding.) So the code in IO is external to MPF, and can illustrate legacy code. To simplify matters, the IO code contains one source file, called `IO.c`, and one header file, called `IO.h`.

The IAC module consists of two Stateflow charts, `spd_filt` and `iac_ctrl`. The `spd_filt` chart has two signals (`meas_spd`) and `filt_spd`, and one parameter (`a`). The `iac_ctrl` chart also has two signals (`filt_spd` and `iac_cmd`) and a parameter (`ref_spd`). (The parameters are not visible in the top-level charts.) One file for each chart is generated. This example system allows us to illustrate referencing from file to file within the MPF module, and model to external module. It also illustrates the case where there is no such referencing.



Proceed to the discussion of the desired example settings:

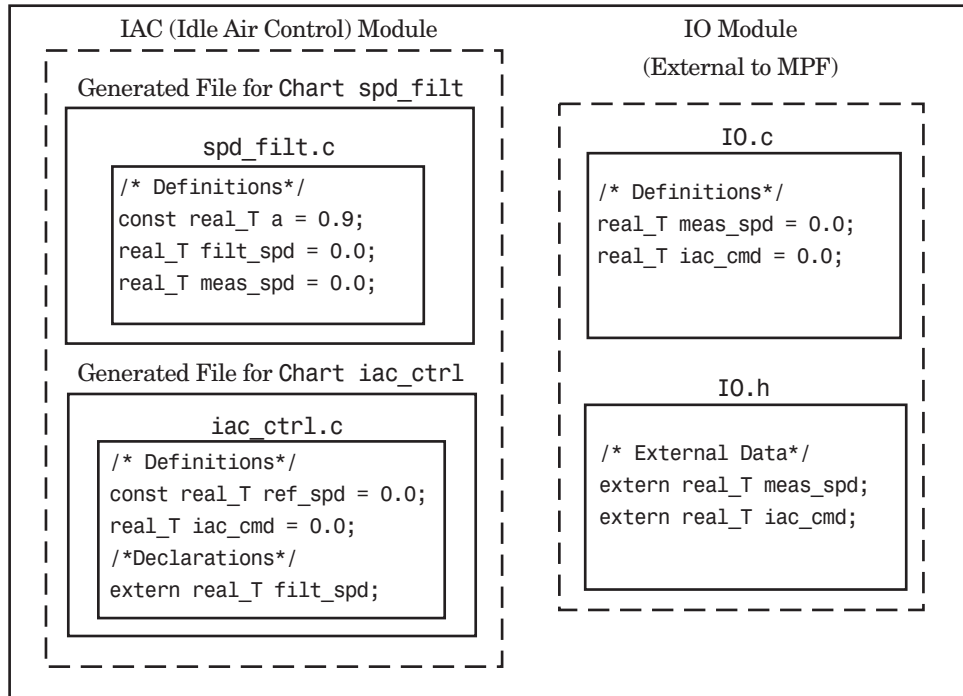
- “Read-Write Example” on page 12-15
- “Ownership Example” on page 12-17
- “Header File Example” on page 12-18
- “Definition File Example” on page 12-20

Read-Write Example

These settings and the generated files that result are shown as Example Settings 1 in “Example Settings and Resulting Generated Files” on page 12-23. As you can see from the table, this example illustrates the case in which only one .c source file (for each chart) is generated.

So, for the IAC model, select the following settings. Accept the Data defined in source file in the **Data definition** field and the Data declared in source file in the **Data declaration** field on the **Data Placement** pane of the Configuration Parameters dialog box. Accept the default Not specified selection in the **Module naming** field. Accept the default blank settings for the **Owner**, **Definition file** and **Header file** fields on the Model Explorer. For **Memory section**, accept Default. Now the Read-Write priority is in

effect. Generate code. The next figure shows the results in terms of definition and declaration statements.



Engine Idle Speed Control System (Read-Write Example)

The code generator generated a `spd_filt.c` for the `spd_filt` chart and `iac_ctrl.c` for the `iac_ctrl` chart. As you can see, MPF placed all definitions of data objects for the `spd_filt` chart in `spd_filt.c`. It placed all definitions of data objects for the `iac_ctrl` chart in `iac_ctrl.c`.

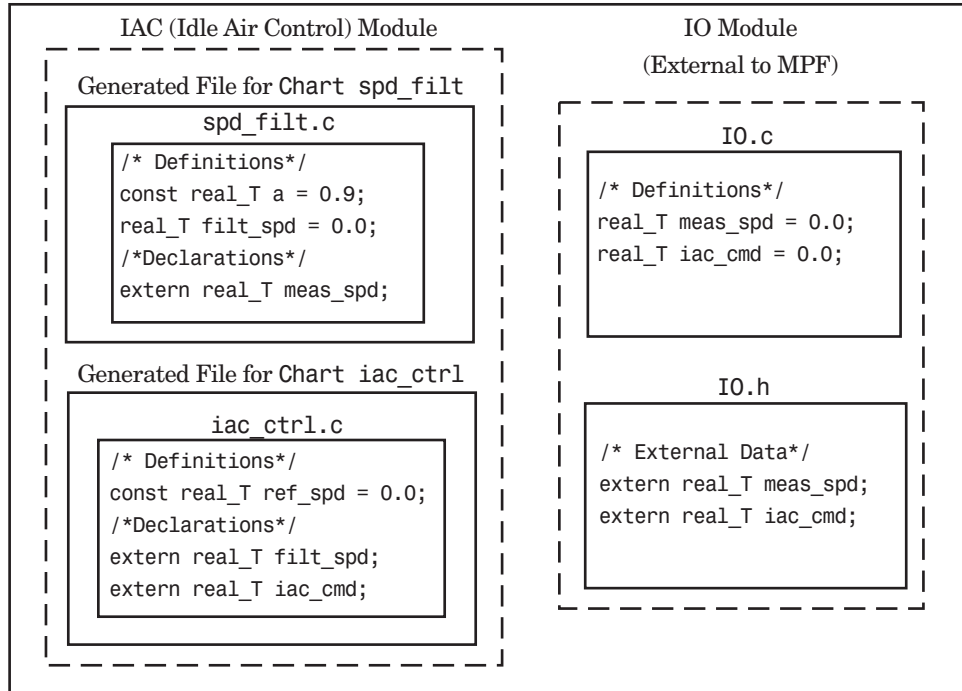
However, notice `real_T filt_spd`. This data object is defined in `spd_filt.c` and declared in `iac_ctrl.c`. That is, since the Read-Write priority is in effect, `filt_spd` is defined in the file that first writes to its address. And, it is declared in the file that reads (uses) it. Further, `real_T meas_spd` is defined in both `spd_filt.c` and the external `IO.c`. And, `real_T iac_cmd` is defined in both `iac_ctrl.c` and `IO.c`.

Ownership Example

See tables “Effects of Ownership Settings” on page 12-22 and “Example Settings and Resulting Generated Files” on page 12-23. In the “Read-Write Example” on page 12-15, there are several instances where the same data object is defined in more than one .c source file, and there is no declaration (extern) statement. This would result in compiler errors during link time. But in this example, we configure MPF Ownership rules so that adequate linking can take place. Notice the Example Settings 2 row in “Example Settings and Resulting Generated Files” on page 12-23. Except for the ownership settings, assume these are the settings you made for the model in the IAC module. Since this example has no **Definition file** or **Header file** specified, now Ownership takes priority. (If there *were* a **Definition file** or **Header file** specified, MPF would ignore the ownership settings.)

On the **Data Placement** pane of the Configuration Parameters dialog box, select User specified in the **Module naming** field, and specify IAC in the **Module name** field (case sensitive). Open the Model Explorer (by issuing the MATLAB command daexplr) and, for all data objects except meas_spd and iac_cmd, type IAC in the **Owner** field (case sensitive). Then, only for the meas_spd and iac_cmd data objects, type IO as their **Owner** (case sensitive). Generate code.

The results are shown in the next figure. Notice the `extern real_T meas_spd` statement in `spd_filt.c`, and `extern real_T iac_cmd` in `iac_ctrl.c`. MPF placed these declaration statements in the correct files where these data objects are used. This allows the generated source files (`spd_filt.c` and `iac_ctrl.c`) to be compiled and linked with `IO.c` without errors.



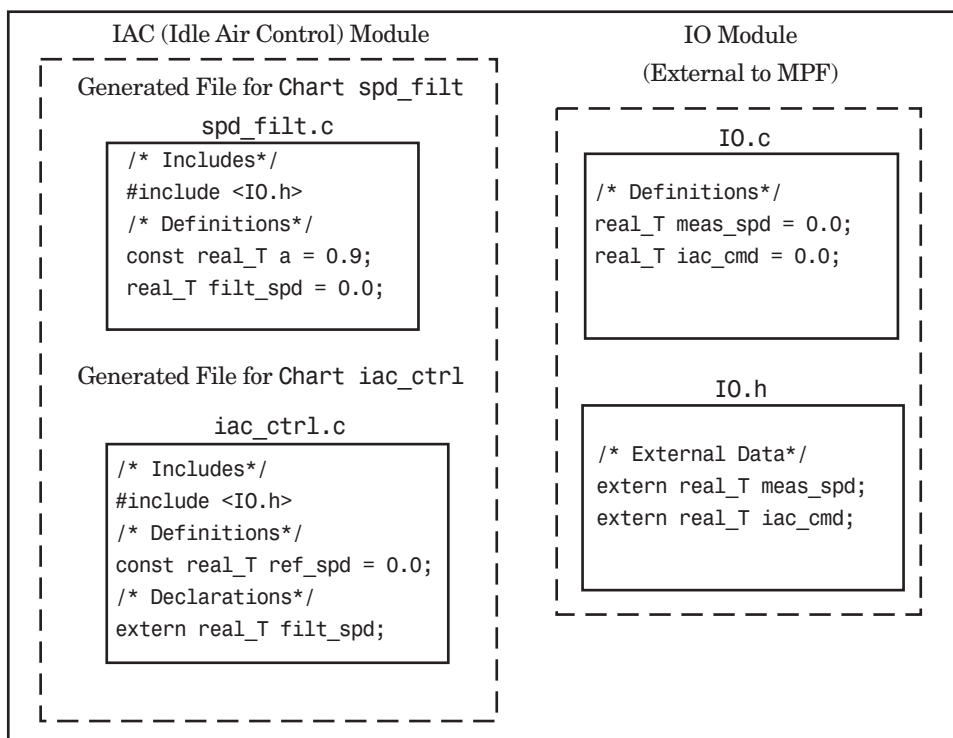
Engine Idle Speed Control System (Ownership Example)

Header File Example

These settings and the generated files that result are shown as Example Settings 3 in “Example Settings and Resulting Generated Files” on page 12-23. Since this example has no **Definition file** specified, it allows us to describe the effects of the **Header file** setting. (If there *were* a **Definition file**, MPF would ignore the **Header file** setting.) The focus of this example is to show how the **Header file** settings result in the linking of the two chart source files to the external IO files, shown in the next figure. (Also, ownership settings will be used to link the two chart files with each other.)

As you can see in the figure, the `meas_spd` and `iac_cmd` identifiers are defined in `IO.c` and declared in `IO.h`. Both of these identifiers are external to the generated `.c` files. You open the Model Explorer and select both the `meas_spd` and `iac_cmd` data objects. For each of these data objects, in the **Header file** field, specify `IO.h`, since this is where these two objects are declared. This setting ensures that the `spd_filt.c` source file will compile and link with the external `IO.c` file without errors.

Now we configure the ownership settings. In the Model Explorer, select the `filt_spd` data object and set its **Owner** field to `IAC`. Then, on the **Data Placement** pane of the Configuration Parameters dialog box, select `User` specified in the **Module naming** field, and specify `IAC` in the **Module Name** field. This ensures that the `spd_filt` source file will link to the `iac_ctrl` source file. Generate code. See the figure below.



Engine Idle Speed Control System (Header File Example)

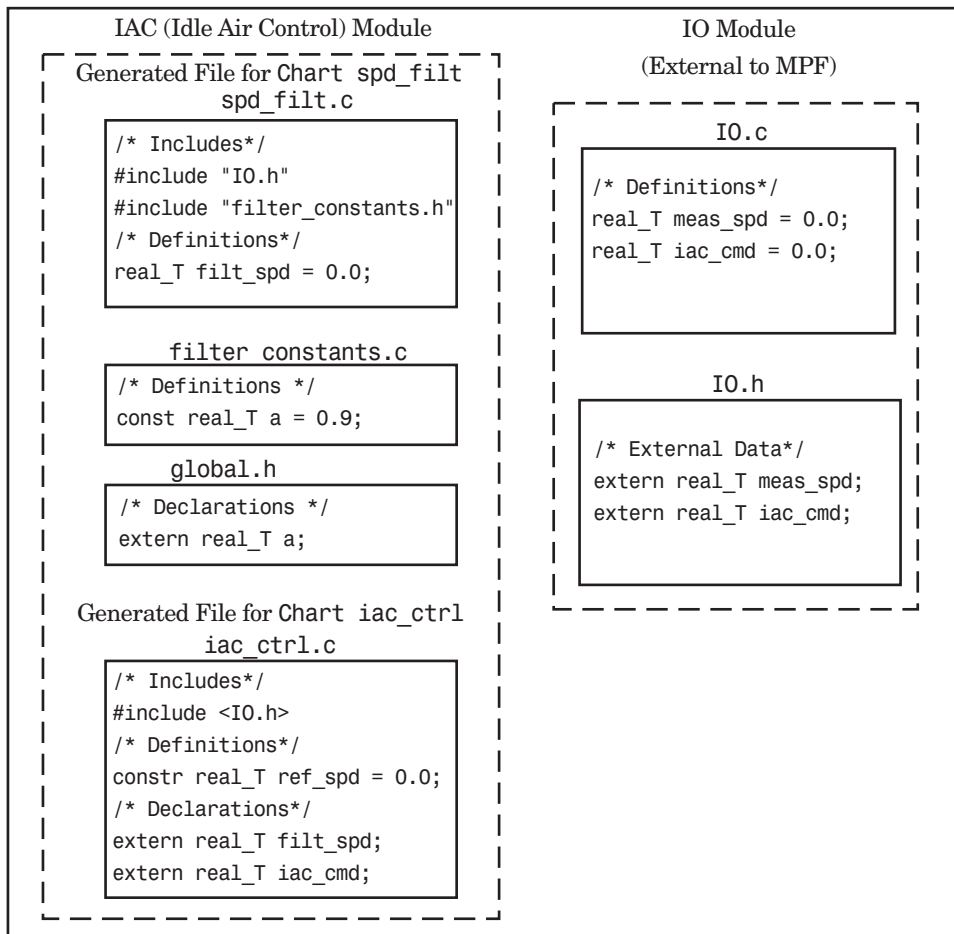
Since you specified the `IO.h` filename for the **Header file** field for the `meas_spd` and `iac_ctrl` objects, the code generator assumed correctly that their declarations are in `IO.h`. So the code generator placed `#include IO.h` in each source file: `spd_filt.c` and `iac_ctrl.c`. So these two files will link with the external `IO` files. Also, due to the ownership settings that were specified, the code generator places the `real_T filt_spd = 0.0;` definition in `spd_filt.c` and declares the `filt_spd` identifier in `iac_ctrl.c` with `extern real_T iac_cmd;`. Consequently, the two source files will link together.

Definition File Example

These settings and the generated files that result are shown as Example Settings 4 in “Example Settings and Resulting Generated Files” on page 12-23. Notice that a definition filename is specified. The settings in the table only apply to the data object called `a`. You have decided that you do not want this object defined in `spd_filt.c`, the generated source file for the `spd_filt` chart. (There are many possible organizational reasons one might want an object declared in another file. It is not important for this example to specify the reason.)

For this example, assume the settings for all data objects are the same as those indicated in “Header File Example” on page 12-18, except for the data object `a`. The description below identifies only the differences that result from this.

Open the Model Explorer, and select data object `a`. In the **Definition file** field you specify any desired filename. Choose `filter_constants.c`. Generate code. The results are shown in the next figure.



Engine Idle Speed Control System (Definition File Example)

The code generator generates the same files as in the “Header File Example” on page 12-18, and adds a new file, `filter_constants.c`. Data object `a` now is defined in `filter_constants.c`, rather than in the source file `spd_filt.c`, as it is in the example. This data object is declared with an `extern` statement in `global.h`.

Data Placement Rules and Effects

In this section...
“Effects of Ownership Settings” on page 12-22
“Example Settings and Resulting Generated Files” on page 12-23
“Data Placement Rules” on page 12-25

Effects of Ownership Settings

Row Number	Module Naming Setting	Owner Setting	Effect*
1	Not specified**	Blank**	There is a definition for the selected data object. The code generator places this definition in the .c/.cpp source file that uses it. There is also an extern declaration for this data object. The code generator places this extern declaration in one or more .h header files, as needed.
2	Not specified**	A name is specified.	Same effect as stated above.
3	Either Same as model or User specified is selected.	Blank**	Same as Row 1.
4	Either Same as model or User specified is selected, and this name is the same as that specified as the Owner property.	A name is specified and it is the same as that specified in the Module naming > Module name field.	Same as Row 1.
5	Either Same as model or User specified is selected, and this name is different than that specified as the Owner property.	A name is specified but it is different from that specified in the Module naming > Module name field.	There is no definition for the selected data object. However, there is an extern declaration for the object. The extern declaration is placed in one or more header files, as needed.

* See also “Ownership Settings” on page 12-10.

** Default.

Example Settings and Resulting Generated Files

	Data Defined In...	Data Declared In...	Ownership*	Defined File**	Header File	Generated Files
Example Settings 1 (Rd-Write Example)	Source file	Source file	Blank	Blank	Blank	.c / .cpp source file
Example Settings 2 (Ownership Example)	Source file	Source file	Name of module specified	Blank	Blank	.c / .cpp source file
Example Settings 3 (Header File Example)	Source file	Source file	Blank	Blank	Desired include filename specified.	.c / .cpp source file .h definition file
Example Settings 4 (Def. File Example)	Source file	Source file	Blank	Desired definition filename specified.	Desired include filename specified.	.c / .cpp source file .c / .cpp definition file* .h definition file*
Example Settings 5	Single separate source file	Source file	Blank	Blank	Blank	.c / .cpp source file global .c / .cpp
Example Settings 6	Single separate source file	Single separate header file	Blank	Blank	Blank	.c / .cpp source file global .c / .cpp global.h

	Data Defined In...	Data Declared In...	Ownership*	Defined File**	Header File	Generated Files
Example Settings 7	Single separate source file	Single separate header file	Name of module specified	Blank	Blank	.c/.cpp source file global.c/.cpp global.h
Example Settings 8	Single separate source file	Single separate header file	Blank	Blank	Desired include filename specified.	.c/.cpp source file global.c/.cpp global.h .h definition file

* "Blank" in ownership setting means Not specified is selected in the **Module naming** field on the **Data Placement** pane, and the **Owner** field on the Model Explorer is blank. "Name of module specified" can be a variety of ownership settings as defined in "Effects of Ownership Settings" on page 12-22.

** The code generator generates a definition .c/.cpp file for every data object for which you specified a definition filename (unless you selected #DEFINE for the **Memory section** field). For example, if you specify the same definition filename for all data objects, only one definition .c/.cpp file is generated. The code generator places declarations in *model.h* by default, unless you specify Data declared in single separate header file for the **Data declaration** option on the **Real-Time Workshop > Data Placement** pane of the Configuration Parameter dialog box. If you select that data placement option, the code generator places declarations in *global.h*. If you specify a definition filename for each data object, the code generator generates one definition .c/.cpp file for each data object and places declarations in *model.h* by default, unless you specify Data declared in single separate header file for **Data declaration**. If you select that data placement option, the code generator places declarations in *global.h*.

Note If you generate C++ rather than C code, the .c files listed in the following table will be .cpp files.

Data Placement Rules

Storage Class Setting	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
<i>mpt or Simulink Noncustom Storage Classes:</i>								
auto	N/A	N/A	N/A	N/A	N/A	Note 12	model.h	Note 1
Exported-Global	N/A	N/A	N/A	N/A	N/A	model.c	model.h	Note 1
Imported-- Extern, Imported-- Extern-Pointer	N/A	N/A	N/A	N/A	N/A	None. External	model_ private.h	Note 2
Simulink-Global	N/A	N/A	N/A	N/A	N/A	Note 13	model.h	Note 1
<i>mpt or Simulink Custom Storage Class: Imported Data:</i>								
Imported-- FromFile	D/C	D/C	D/C	N/A	null	None	model_ private.h	Note 3
Imported-- FromFile	D/C	D/C	D/C	N/A	hdr.h	None	model_ private.h	Note 4
<i>Simulink Custom Storage Class: #define Data:</i>								
Define	D/C	D/C	N/A	N/A	N/A	N/A	#define, model.h	Note 5
<i>mpt Custom Storage Class: #define Data:</i>								
Define	D/C	D/C	N/A	N/A	null	N/A	#define, model.h	Note 5
Define	D/C	D/C	N/A	N/A	hdr.h	N/A	#define, model.h	Note 6
<i>mpt or Simulink Custom Storage Class: GetSet:</i>								
GetSet	D/C	D/C	N/A	N/A	hdr.h	N/A	External hdr.h	Note 4

Storage Class Setting	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
<i>mpt or Simulink Custom Storage Class: Bitfield, Struct:</i>								
Bitfield, Struct	D/C	D/C	N/A	N/A	N/A	model.c	model.h	Note 7
<i>mpt Custom Storage Class: Global, Const, ConstVolatile, Volatile:</i>								
Global, Const, Const-Volatile, Volatile	auto	auto	null	null or locally owned	null	model.c	model.h	Note 1
Global, Const, Const-Volatile, Volatile	src	auto	null	null or locally owned	null	src.c	model.h	Note 1
Global, Const, Const-Volatile, Volatile	sep	auto	null	null or locally owned	null	gbl.c	model.h	Note 1
Global, Const, Const-Volatile, Volatile	auto	src	null	null or locally owned	null	model.c	src.c	Note 8
Global, Const, Const-Volatile, Volatile	src	src	null	null or locally owned	null	src.c	src.c	Note 8
Global, Const, Const-Volatile, Volatile	sep	src	null	null or locally owned	null	gbl.c	src.c	Note 8
Global, Const, Const-Volatile, Volatile	auto	sep	null	null or locally owned	null	model.c	gbl.h	Note 9
Global, Const, Const-Volatile, Volatile	src	sep	null	null or locally owned	null	src.c	gbl.h	Note 9
Global, Const, Const-Volatile, Volatile	sep	sep	null	null or locally owned	null	gbl.c	gbl.h	Note 9
Global, Const, Const-Volatile, Volatile	D/C	D/C	data.c	D/C	null	data.c	See Note 10.	Note 10

Storage Class Setting	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
Global, Const, Const-Volatile, Volatile	D/C	D/C	data.c	D/C	hdr.h	data.c	hdr.h	Note 11
Global, Const, Const-Volatile, Volatile	auto	D/C	null	null	hdr.h	model.c	hdr.h	Note 11
Global, Const, Const-Volatile, Volatile	src	D/C	null	null	hdr.h	src.c	hdr.h	Note 11
Global, Const, Const-Volatile, Volatile	sep	D/C	null	null	hdr.h	gbl.c	hdr.h	Note 11
Global, Const, Const-Volatile, Volatile	D/C	auto	null	External owner	null	External user--supplied file	model.h	Note 1
Global, Const, Const-Volatile, Volatile	D/C	src	null	External owner	null	External user--supplied file	src.c	Note 8
Global, Const, Const-Volatile, Volatile	D/C	sep	null	External owner	null	External user--supplied file	gbl.h	Note 9
Global, Const, Const-Volatile, Volatile	D/C	D/C	null	External owner	header.h	External user--supplied file	hdr.h	Note 11
Global, Const, Const-Volatile, Volatile	D/C	D/C	null	External owner	header.h	External user--supplied file	hdr.h	Note 11

mpt Custom Storage Class: Exported Data:

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
ExportTo-File	auto	auto	null	null	null	model.c	model.h	Note 1
ExportTo-File	src	auto	null	null	null	src.c	model.h	Note 1
ExportTo-File	sep	auto	null	null	null	gbl.c	model.h	Note 1
ExportTo-File	auto	src	null	null	null	model.c	src.c	Note 8
ExportTo-File	src	src	null	null	null	src.c	src.c	Note 8
ExportTo-File	sep	src	null	null	null	gbl.c	src.c	Note 8
ExportTo-File	auto	sep	null	null	null	model.c	gbl.h	Note 9
ExportTo-File	src	sep	null	null	null	src.c	gbl.h	Note 9
ExportTo-File	sep	sep	null	null	null	gbl.c	gbl.h	Note 9
ExportTo-File	D/C	D/C	data.c	null	null	data.c	See Note 10.	Note 10
ExportTo-File	D/C	D/C	data.c	null	hdr.h	model.c	hdr.h	Note 11
ExportTo-File	auto	D/C	null	null	hdr.h	src.c	hdr.h	Note 11
ExportTo-File	sep	D/C	null	null	hdr.h	gbl.c	hdr.h	Note 11
<i>Simulink Custom Storage Class: Default, Const, ConstVolatile, Volatile:</i>								
Default, Const, Const-Volatile, Volatile	auto	auto	N/A	N/A	N/A	model.c	model.h	Note 1
Default, Const, Const-Volatile, Volatile	src	auto	N/A	N/A	N/A	src.c	model.h	Note 1
Default, Const, Const-Volatile, Volatile	sep	auto	N/A	N/A	N/A	gbl.c	model.h	Note 1
Default, Const, Const-Volatile, Volatile	auto	src	N/A	N/A	N/A	model.c	src.c	Note 8
Default, Const, Const-Volatile, Volatile	src	src	N/A	N/A	N/A	src.c	src.c	Note 8

Storage Class Setting	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion
Default, Const, Const-Volatile, Volatile	sep	src	N/A	N/A	N/A	gbl.c	src.c	Note 8
Default, Const, Const-Volatile, Volatile	auto	sep	N/A	N/A	N/A	model.c	gbl.h	Note 9
Default, Const, Const-Volatile, Volatile	src	sep	N/A	N/A	N/A	src.c	gbl.h	Note 9
Default, Const, Const-Volatile, Volatile	sep	sep	N/A	N/A	N/A	gbl.c	gbl.h	Note 9
<i>Simulink Custom Storage Class: Exported Data:</i>								
ExportTo-File	auto	auto	N/A	N/A	null	model.c	model.h	Note 1
ExportTo-File	src	auto	N/A	N/A	null	src.c	model.h	Note 1
ExportTo-File	sep	auto	N/A	N/A	null	gbl.c	model.h	Note 1
ExportTo-File	auto	src	N/A	N/A	null	model.c	src.c	Note 8
ExportTo-File	src	src	N/A	N/A	null	src.c	src.c	Note 8
ExportTo-File	sep	src	N/A	N/A	null	gbl.c	src.c	Note 8
ExportTo-File	auto	sep	N/A	N/A	null	model.c	gbl.h	Note 9
ExportTo-File	src	sep	N/A	N/A	null	src.c	gbl.h	Note 9
ExportTo-File	sep	sep	N/A	N/A	null	gbl.c	gbl.h	Note 9
ExportTo-File	auto	D/C	N/A	N/A	hdr.h	model.c	hdr.h	Note 11
ExportTo-File	src	D/C	N/A	N/A	hdr.h	src.c	hdr.h	Note 11
ExportTo-File	sep	D/C	N/A	N/A	hdr.h	gbl.c	hdr.h	Note 11

Notes

In the previous table:

- A Declaration Inclusion Approach is a file in which the header file that contains the data declarations is included.
- D/C stands for don't care.
- Dec stands for declaration.
- Def stands for definition.
- gbl stands for global.
- hdr stands for header.
- N/A stands for not applicable.
- null stands for field is blank.
- sep stands for separate.

Note 1: `model.h` is included directly in all source files.

Note 2: `model_private.h` is included directly in all source files.

Note 3: `extern` is included in `model_private.h`, which is in `source.c`.

Note 4: `header.h` is included in `model_private.h`, which is in `source.c`.

Note 5: `model.h` is included directly in all source files that use `#define`.

Note 6: `header.h` is included in `model.h`, which is in source files that use `#define`.

Note 7: `model.h` is included in all `source.c` files.

Note 8: `extern` is inlined in source files where data is used.

Note 9: `global.h` is included in `model.h`, which is in all source files.

Note 10: When you specify a definition filename for a data object, no header file is generated for that data object. The code generator declares the data object according to the data placement priorities.

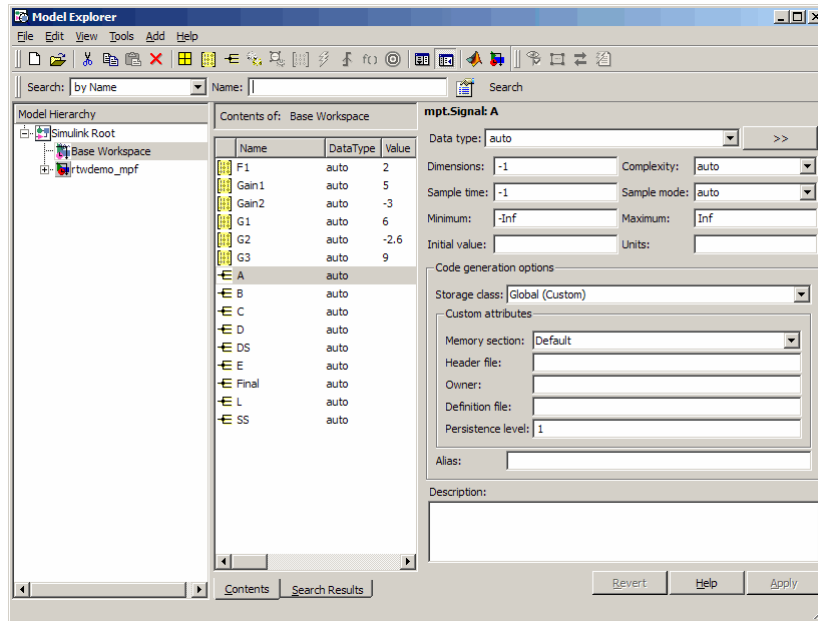
Note 11: `header.h` is included in `model.h`, which is in all source files.

Note 12: Signal: Either not defined because it is expression folded, or local data, or defined in a structure in `model.c`, all depending on model's code generation settings. Parameter: Either inlined in the code, or defined in `model_data.c`.

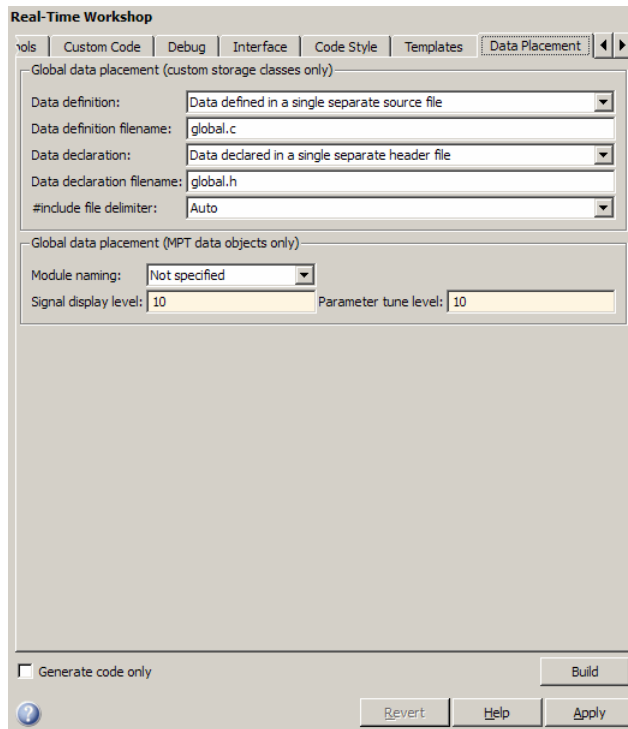
Note 13: Signal: In a structure that is defined in `model.c`. Parameter: In a structure that is defined in `model_data.c`.

Specifying the Persistence Level for Signals and Parameters

With this procedure, you can control the persistence level of signal and parameter objects associated with a model. Persistence level allows you to make intermediate variables or parameters global during initial development. At the later stages of development, you can use this procedure to remove these signals and parameters for efficiency. Notice the **Persistence Level** field on the Model Explorer, as illustrated in the figure below. For descriptions of the properties on the Model Explorer, see Parameter and Signal Property Values on page 6-2.



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



The **Signal display level** field allows you to specify whether or not the code generator defines a signal data object as global data in the generated code. The number you specify in this field is relative to the number you specify in the **Persistence level** field. The **Signal display level** number is for all mpt (module packaging tool) signal data objects in the model. The **Persistence level** number is for a *particular* mpt signal data object. If the data object's **Persistence level** is equal to or less than the **Signal display level**, the signal appears in the generated code as global data with all of the properties (custom attributes) specified in "Creating mpt Data Objects with Data Object Wizard" on page 11-11. For example, this would occur if **Persistence level** is 2 and **Signal display level** is 5.

Otherwise, the code generator automatically determines how the particular signal data object appears in the generated code. Depending on the settings on the **Optimization** pane of the Configuration Parameters dialog box, the signal data object could appear in the code as local data and thus have none

of the custom attributes you specified for that data object. Or, based on expression folding, the code generator could remove the data object so that it does not appear in the code. (See “Tips for Optimizing the Generated Code” on page 20-19 in the Real-Time Workshop Embedded Coder documentation and “Optimizing Generated Code” in the Real-Time Workshop documentation for details on optimization.)

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

The number you specify in this field is relative to the number you specify in the **Persistence level** field. The **Parameter tune level** number is for all mpt parameter data objects in the model. The **Persistence level** number is for a *particular* mpt parameter data object. If the data object’s **Persistence level** is equal to or less than the **Parameter tune level**, the parameter appears in the generated code with all of the properties (custom attributes) specified in “Creating mpt Data Objects with Data Object Wizard” on page 11-11, and thus is tunable. For example, this would occur if **Persistence level** is 2 and **Parameter tune level** is 5.

Otherwise, the parameter is inlined in the generated code, and Real-Time Workshop settings determine its exact form.

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

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

Preparing Models for Code Generation

- Chapter 14, “Mapping Application Objectives to Model Configuration Parameters”
- Chapter 15, “Choosing and Configuring an Embedded Real-Time Target”
- Chapter 16, “Specifying Code Appearance and Documentation”
- Chapter 17, “Defining Model Configuration Variations”

Mapping Application Objectives to Model Configuration Parameters

- “Considerations When Mapping Application Objectives” on page 14-2
- “Defining High-Level Code Generation Objectives” on page 14-3
- “Determining Whether the Model is Configured for Specified Objectives” on page 14-4
- “Creating Custom Objectives” on page 14-11

Considerations When Mapping Application Objectives

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

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

Consider questions such as the following:

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

After you answer these questions:

- 1** Define your objectives in the configuration set. For more information, see “Defining High-Level Code Generation Objectives” on page 14-3.
- 2** Use the Code Generation Advisor to identify parameter values that are not configured for the objectives that you selected. For more information, see “Determining Whether the Model is Configured for Specified Objectives” on page 14-4.

Defining High-Level Code Generation Objectives

When you are considering the objectives for your application, there are many different criteria. The Real-Time Workshop software identifies six high-level objectives that you might consider for your application:

- Execution efficiency — Configure code generation settings to achieve fast execution time.
- ROM efficiency — Configure code generation settings to reduce ROM usage.
- RAM efficiency — Configure code generation settings to reduce RAM usage.
- Traceability — Configure code generation settings to provide mapping between model elements and code.
- Safety precaution — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.
- Debugging — Configure code generation settings to debug the code generation build process.

Once you have identified which of these four objectives are important for your application, you can use the Code Generation Advisor to identify the parameters that are not configured for the objectives that you selected. Review “Recommended Settings Summary” to see the settings the Code Generation Advisor recommends.

You can specify and prioritize any combination of the available objectives for the Code Generation Advisor to take into consideration. For more information, see “Determining Whether the Model is Configured for Specified Objectives” on page 14-4.

Determining Whether the Model is Configured for Specified Objectives

In this section...
“Specifying Code Generation Objectives Using the GUI” on page 14-4
“Specifying Code Generation Objectives at the Command Line” on page 14-6
“Reviewing Objectives in Referenced Models” on page 14-7
“Reviewing the Model Without Generating Code” on page 14-7
“Reviewing the Model During Code Generation” on page 14-9

You can use the Code Generation Advisor to review your model and identify the parameters that are not configured for your objective. The Code Generation Advisor reviews a subset of model configuration parameters and displays the results in the **Check model configuration settings against code generation objectives** check.

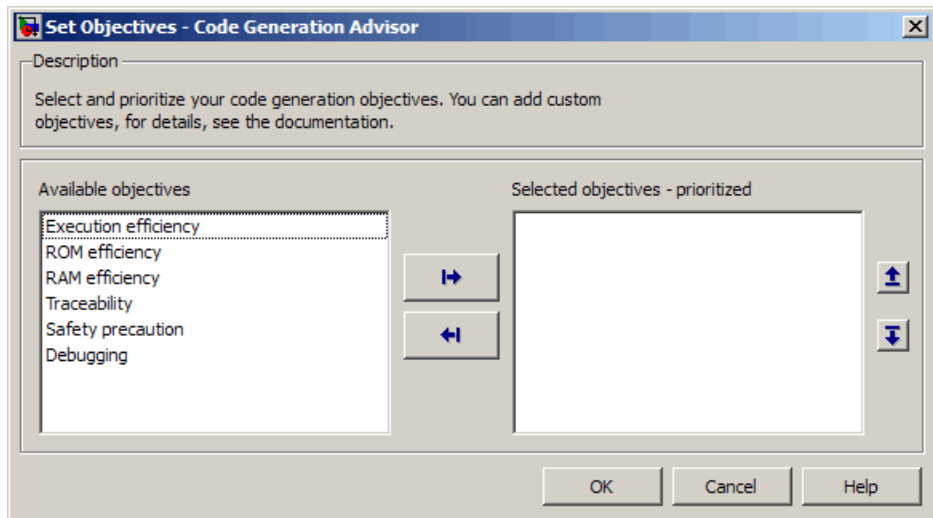
The Code Generation Advisor uses the information presented in “Mapping of Application Requirements to the Optimization Pane” to determine the recommended values. When there is a conflict due to multiple objectives, the higher-priority objective takes precedence.

Tip You can use the Code Generation Advisor to review a model before generating code, or as part of the code generation process. When you choose to review a model before generating code, you specify which model, subsystem, or referenced model the Code Generation Advisor reviews (see “Reviewing the Model Without Generating Code” on page 14-7). When you choose to review a model as part of the code generation process, the Code Generation Advisor reviews the entire system (see “Reviewing the Model During Code Generation” on page 14-9).

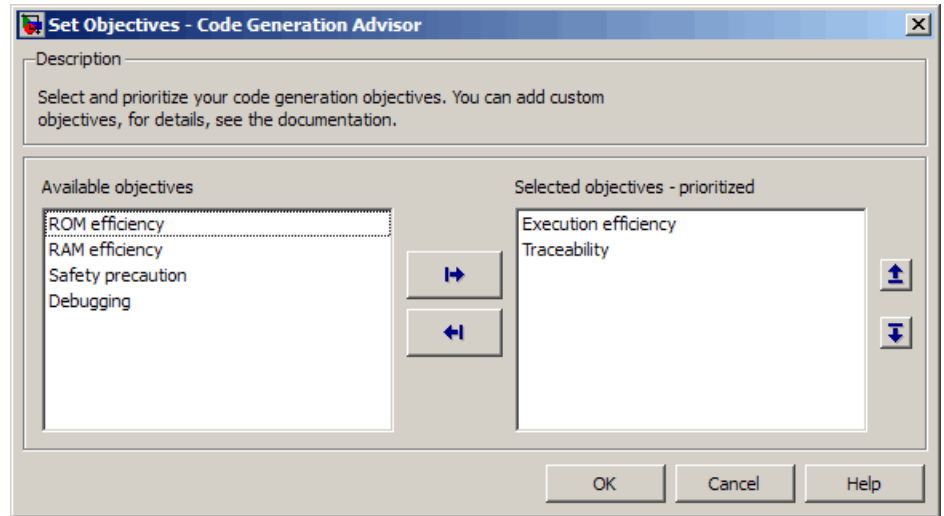
Specifying Code Generation Objectives Using the GUI

To specify code generation objectives in the Configuration Parameters dialog box:

- 1 Open the Configuration Parameters dialog box and select the **Real-Time Workshop** pane.
- 2 Specify a system target file. If you specify an ERT-based target, more objectives are available. For the purposes of this example, choose an ERT-based target such as `ert.tlc`.
- 3 Click **Set objectives**. The Configuration Set Objectives dialog box opens.



- 4 In the Configuration Set Objectives dialog box, specify your objectives. For example, if your objectives are efficiency and traceability, in that priority, do the following:
 - a In **Available objectives**, double-click Execution efficiency. Execution efficiency is added to **Prioritized objectives**.
 - b In **Available objectives**, double-click Traceability. Traceability is added to **Prioritized objectives** below Execution efficiency.



- c Click **OK** to accept the objectives. In the Configuration Parameters dialog box, **Prioritized objectives** is updated.

Specifying Code Generation Objectives at the Command Line

To specify code generation objectives by writing a MATLAB script or entering commands at the command line:

- 1 Specify a system target file. If you specify an ERT-based target, more objectives are available. For the purposes of this example, specify `ert.tlc`, where *model_name* is the name or handle to the model.

```
set_param(model_name, 'SystemTargetFile', 'ert.tlc');
```

- 2 Specify your objectives. For example, if your objectives are execution efficiency and traceability, in that priority, enter:

```
set_param(model_name, 'ObjectivePriorities', ...
{'Execution efficiency', 'Traceability'});
```

Caution When you specify a GRT-based system target file, you can specify any objective at the command line. If you specify Execution efficiency, ROM efficiency, RAM efficiency, Traceability, or Safety precaution, the build process changes the objective to Unspecified because you have specified a value that is invalid when using a GRT-based target.

Reviewing Objectives in Referenced Models

When you review a model during the code generation process, you must specify the same objectives in the top model and referenced models. If you specify different objectives for the top model and referenced model, the build process generates an error.

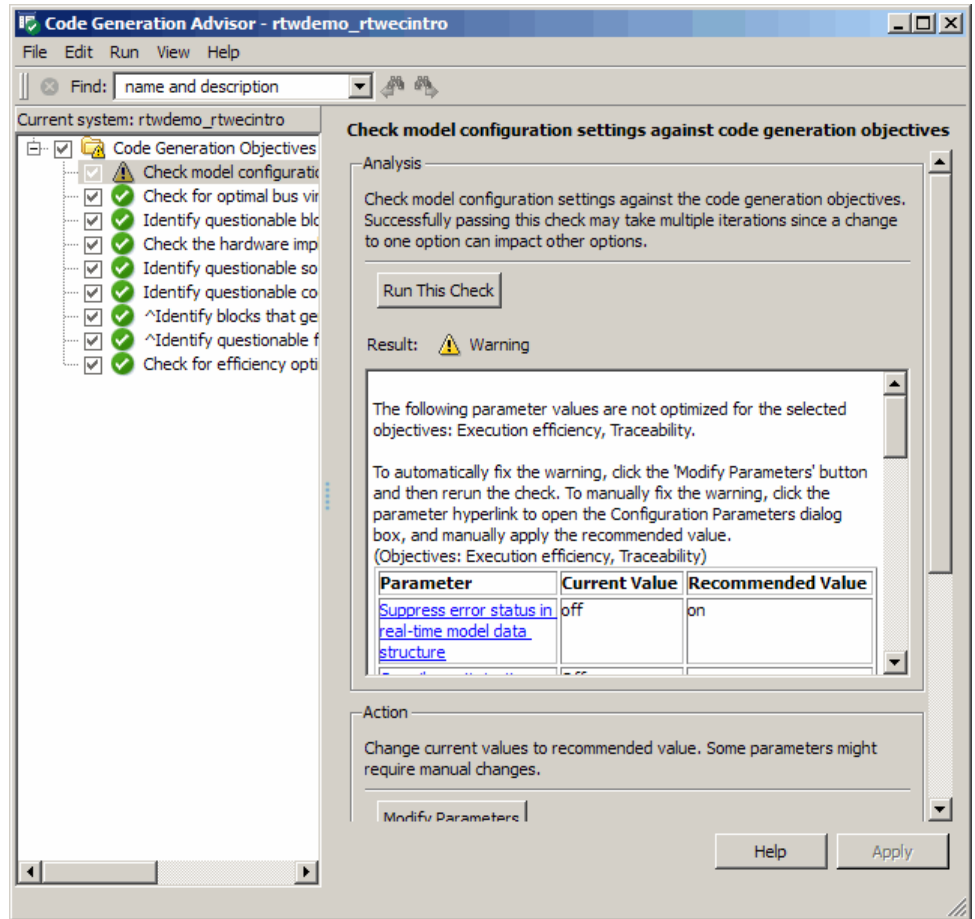
To specify different objectives for the top model and each referenced model, review the models separately without generating code.

Reviewing the Model Without Generating Code

To review a model without generating code using the Code Generation Advisor:

- 1 Specify your code generation objectives.
- 2 In the **Configuration Parameters > Real-Time Workshop** pane, click **Check model**. The System Selector window opens.
- 3 Select the model or subsystem that you want to review and click **OK**. The Code Generation Advisor opens and reviews the model or subsystem that you specified.

- 4 In the Code Generation Advisor window, review the results by selecting a check from the left pane. The right pane populates the results for that check.



- After reviewing the check results, you can choose to fix warnings and failures, as described in “Fixing a Warning or Failure” in the *Simulink User’s Guide*.

Caution When you specify an efficiency or safety precaution objective, the Code Generation Advisor includes additional checks.

When you make changes to one check, the other check results are no longer valid and you must run the checks again for accurate results.

Reviewing the Model During Code Generation

To review a model as part of the code generation process using the Code Generation Advisor:

- 1 Specify your code generation objectives.
- 2 In the **Configuration Parameters > Real-Time Workshop** pane, select one of the following from **Check model before generating code**:
 - On (proceed with warnings)
 - On (stop for warnings)
- 3 Select **Generate code only** if you only want to generate code; otherwise clear the check box to build an executable.
- 4 Apply your changes and then click **Generate code/Build**. The Code Generation Advisor starts and reviews the top model and subsystems.

If there are no failures or warnings in the Code Generation Advisor, the build process proceeds. If there are failures or warnings and you specified:

- On (proceed with warnings) — The Code Generation Advisor window opens while the build process proceeds. You can review the results after the build process is complete.
 - On (stop for warnings) — The build process halts and displays the diagnostics viewer. To continue, you must review and resolve the Code Generation Advisor results or change the **Check model before generating code** selection.
- 5 In the Code Generation Advisor window, review the results by selecting a check from the left pane. The right pane populates the results for that check.

- 6 After reviewing the check results, you can choose to fix warnings and failures as described in “Fixing a Warning or Failure” in the *Simulink User’s Guide*.

Caution When you specify an efficiency or safety precaution objective, the Code Generation Advisor includes additional checks. When you make changes to one of these checks, the other check results are no longer valid and you must run the check again for accurate results.

Creating Custom Objectives

In this section...

“Specifying Parameters in Custom Objectives” on page 14-11

“Specifying Checks in Custom Objectives” on page 14-12

“Determining Checks and Parameters in Existing Objectives” on page 14-12

“How to Create Custom Objectives” on page 14-14

The Code Generation Advisor reviews your model based on objectives that you specify. If the predefined efficiency, traceability, safety precaution, and debugging objectives do not meet your requirements, you can create custom objectives.

You can create custom objectives by:

- Creating a new objective and adding parameters and checks to a new objective.
- Creating a new objective based on an existing objective, then adding, modifying and removing the parameters and checks within the new objective.

Specifying Parameters in Custom Objectives

When you create a custom objective, you specify the values of configuration parameters that the Code Generation Advisor reviews using the following methods:

- `addParam` — Add parameters and specify the values that the Code Generation Advisor reviews in **Check model configuration settings against code generation objectives**. When you add parameters that have dependencies, the software includes the dependencies in the list of parameter values that the Code Generation Advisor reviews.
- `modifyInheritedParam` — Modify inherited parameter values that the Code Generation Advisor reviews in **Check model configuration settings against code generation objectives**.

- `removeInheritedParam` — Remove inherited parameters from a new objective that is based on an existing objective. When a user selects multiple objectives, if another selected objective includes this parameter, the Code Generation Advisor reviews the parameter value in **Check model configuration settings against code generation objectives**.

Specifying Checks in Custom Objectives

All objectives include the **Check model configuration settings against code generation objectives** check by default. When you create a custom objective, you specify the list of additional checks that are associated with the custom objective using the following methods:

- `addCheck` — Add checks to the Code Generation Advisor. When a user selects the custom objective, the Code Generation Advisor displays the check, unless the user specifies an additional objective with a higher priority that excludes the check.

For example, you might add a check to the Code Generation Advisor to include a custom check in the automatic model checking process.

- `excludeCheck` — Exclude checks from the Code Generation Advisor. When a user selects multiple objectives, if the user specifies an additional objective that includes this check as a higher priority objective, the Code Generation Advisor displays this check.

For example, you might exclude a check from the Code Generation Advisor when a check takes a long time to process.

- `removeInheritedCheck` — Remove inherited checks from a new objective that is based on an existing objective. When a user selects multiple objectives, if another selected objective includes this check, the Code Generation Advisor displays the check.

For example, you might remove an inherited check, rather than exclude the check, when the check takes a long time to process, but the check is important for another objective.

Determining Checks and Parameters in Existing Objectives

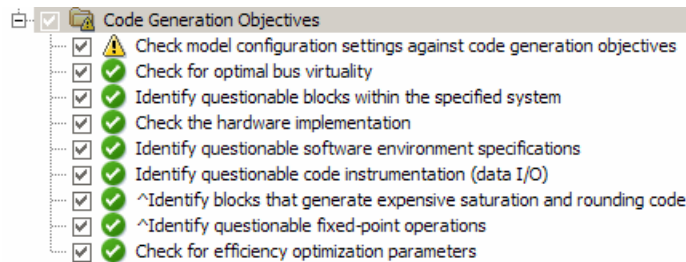
When you base a new objective on an existing objective, you can determine what checks and parameters the existing objective contains. Select the

existing objective and check the model. The Code Generation Advisor lists the checks and parameters associated with the existing objective.

The Code Generation Advisor contains the list of checks in each objective. For example, the `Efficiency` objective includes ten checks, which you can see in the Code Generation Advisor if you:

- 1 Open the `rtwdemo_rtweccintro` model.
- 2 Specify an ERT-based target.
- 3 Specify the `Execution efficiency` objective.
- 4 Check the model.

The Code Generation Advisor displays in the right pane the list of checks in the `Execution efficiency` objective.



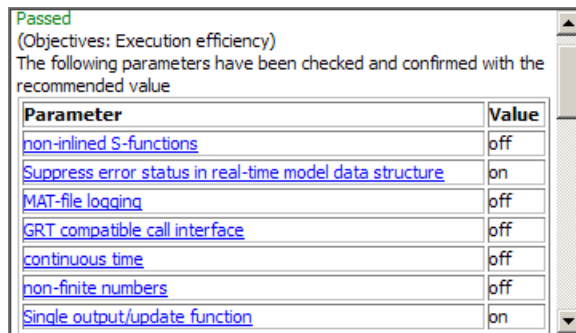
Caution The following objectives *exclude* the listed checks. For more information about excluding checks, see `excludeCheck`.

Objective	Excluded Checks
Traceability	<ul style="list-style-type: none"> • Identify questionable software environment specifications • Identify questionable code instrumentation (data I/O)
Debugging	Identify questionable code instrumentation (data I/O)

The first check, **Check model configuration settings against code generation objectives**, lists all parameters and values specified by the objective. For example, the Code Generation Advisor displays the list of parameters and the recommended values in the Execution efficiency objective, if you:

- 1 Run **Check model configuration settings against code generation objectives**.
- 2 Click **Modify Parameters**.
- 3 Rerun the check.

The Code Generation Advisor displays in the check results the list of parameters and recommended values in the Execution efficiency objective.



Passed
(Objectives: Execution efficiency)
The following parameters have been checked and confirmed with the recommended value

Parameter	Value
non-inlined S-functions	off
Suppress error status in real-time model data structure	on
MAT-file logging	off
GRT compatible call interface	off
continuous time	off
non-finite numbers	off
Single output/update function	on

How to Create Custom Objectives

To create a custom objective:

- 1 Create an `sl_customization.m` file.

Note

- Specify all custom objectives in a single `sl_customization.m` file only, or the software generates an error. This holds true even if you have more than one `sl_customization.m` file on your MATLAB path.
 - Except for the `matlabroot/work` folder, do not place an `sl_customization.m` file in your root MATLAB folder, or any of its subfolders. Otherwise, the software ignores the customizations that the file specifies.
-

- 2** Create an `sl_customization` function that takes a single argument. When the software invokes the function, the value of this argument is the Simulink customization manager. In the function:
 - a** Create a handle to the code generation objective, using the `ObjectiveCustomizer` constructor.
 - b** Register a callback function for the custom objectives, using the `ObjectiveCustomizer.addCallbackObjFcn` method.
 - c** Add a call to execute the callback function, using the `ObjectiveCustomizer.callbackFcn` method.

For example

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn(index)();

end
```

- 3** Create a MATLAB callback function that:
 - Creates code generation objective objects using the `rtw.codegenObjectives.Objective` constructor.

- Adds, modifies, and removes configuration parameters for each objective using the `addParam`, `modifyInheritedParam`, and `removeInheritedParam` methods.
- Includes and excludes checks for each objective using the `addCheck`, `excludeCheck`, and `removeInheritedCheck` methods.
- Registers objectives using the `register` method.

The following example shows you how to create an objective, `Reduce RAM Example`. `Reduce RAM Example` includes five parameters and three checks that the Code Generation Advisor reviews:

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'InlineParams', 'on');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'Identify unconnected lines, input ports, and output ports');
addCheck(obj, 'Check model and local libraries for updates');

% Register the objective
register(obj);

end
```

The following example shows you how to create an objective, `My Traceability Example`, based on the existing `Traceability` objective. The custom objective modifies, removes, and adds parameters that the Code

Generation Advisor reviews. It also adds and removes checks from the Code Generation Advisor:

```
function addObjectives

% Create the custom objective from an existing objective
obj = rtw.codegenObjectives.Objective('ex_my_trace_1', 'Traceability');
setObjectiveName(obj, 'My Traceability Example');

% Modify parameters in the objective
modifyInheritedParam(obj, 'GenerateTraceReportSf', 'Off');
removeInheritedParam(obj, 'ConditionallyExecuteInputs');
addParam(obj, 'MatFileLogging', 'On');

% Modify checks in the objective
addCheck(obj, 'Identify questionable software environment specifications');
removeInheritedCheck(obj, 'Identify questionable code instrumentation (data I/O)');

%Register the objective
register(obj);

end
```

- 4** If you previously opened the Code Generation Advisor, close the model from which you opened the Code Generation Advisor.
- 5** Refresh the customization manager. At the MATLAB command line, enter the `sl_refresh_customizations` command.
- 6** Open your model and review the new objectives.

Choosing and Configuring an Embedded Real-Time Target

- “Introduction” on page 15-2
- “Selecting an ERT Target” on page 15-4
- “Selecting a System Target File” on page 15-5
- “Selecting a System Target File Programmatically” on page 15-7
- “Customizing an ERT Target” on page 15-8

Introduction

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 System Target File” on page 15-5). 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 15, “Choosing and Configuring an Embedded Real-Time 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 System Target File” on page 15-5. 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.

Selecting an ERT Target

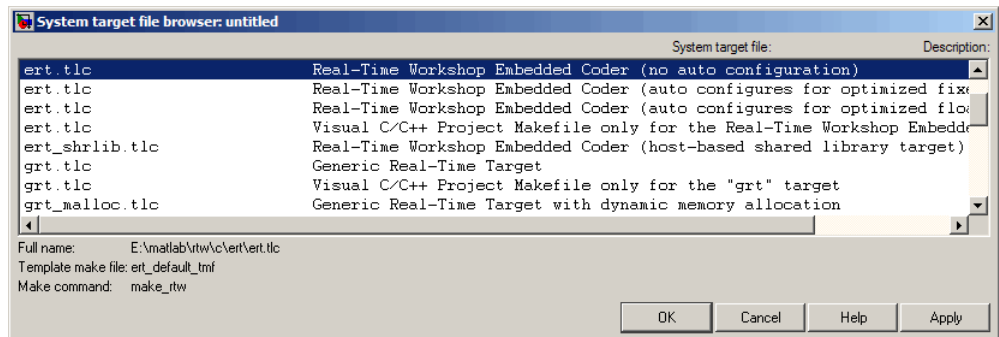
The **Browse** button in the **Target Selection** subpane of the **Real-Time Workshop > General** pane lets you select an ERT target with the System Target File Browser. See “Selecting and Configuring a Target ” in the Real-Time Workshop documentation for a general discussion of target selection.

To make it easier for you to generate code that is optimized for your target hardware, the code generator provides three variants of the ERT target that

- Automatically configure parameters that are optimized for fixed-point code generation
- Automatically configure parameters that are optimized for floating-point code generation
- Applies default parameter settings

The discussion throughout this chapter assumes use of the default ERT target.

These targets are based on a common system target file, `ert.tlc`. They are displayed in the System Target File Browser as shown in the figure below.



You can use the `ert_shrlib.tlc` target to generate a host-based shared library from your Simulink model. Selecting this target allows you to generate a shared library version of your model code that is appropriate for your host platform, either a Windows dynamic link library (.dll) file or a UNIX shared object (.so) file. This feature can be used to package your source code securely

for easy distribution and shared use. For more information, see “Creating and Using Host-Based Shared Libraries” on page 3-9.

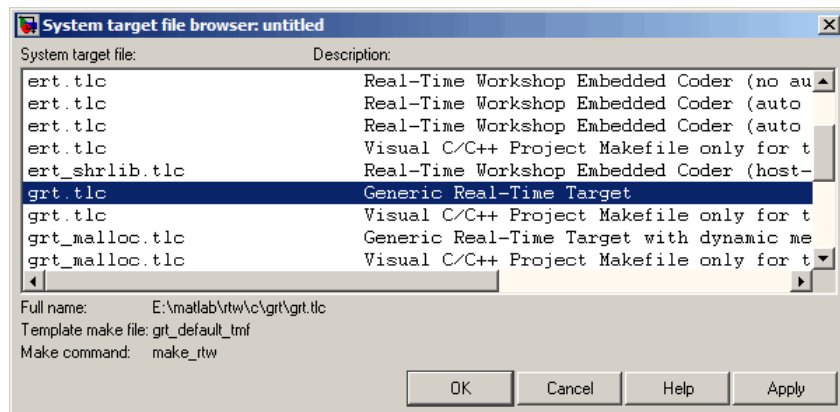
Selecting a System Target File

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 System Target File” on page 15-5 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);
```

Customizing an ERT Target

For information on customizing ERT targets, see “Customizing Targets” in the Real-Time Workshop documentation.

Specifying Code Appearance and Documentation

- “Customizing Comments in Generated Code” on page 16-2
- “Configuring the Appearance of Generated Identifiers” on page 16-11
- “Controlling Code Style” on page 16-21
- “Configuring Templates for Customizing Code Organization and Format” on page 16-22
- “Configuring the Placement of Data in Generated Code” on page 16-67
- “Ensuring Delimiter Is Specified for All #Includes” on page 16-68

Customizing Comments in Generated Code

In this section...

“Adding Custom Comments to Generated Code” on page 16-2

“Adding Global Comments” on page 16-5

Adding Custom Comments to Generated Code

You can customize the comments in the generated code for ERT targets by setting or clearing several parameters on the **Real-Time Workshop > Comments** pane. These options let you enable or suppress generation of descriptive information in comments for blocks and other objects in the model.

To...	Select...
Include the text specified in the Description field of a block's Block Properties dialog box as comments in the code generated for each block	Simulink block descriptions.
Add a comment that includes the block name at the start of the code for each block	Simulink block descriptions
Include the text specified in the Description field of a Simulink data object (such as a signal, parameter, data type, or bus) in the Simulink Model Explorer as comments in the code generated for each object	Simulink data object descriptions.
Include comments just above signals and parameter identifiers in the generated code as specified in the MATLAB or TLC function.	Custom comments (MPT objects only).

To...	Select...
Include the text specified in the Description field of the Properties dialog box for a Stateflow object as comments just above the code generated for each object	Stateflow object descriptions .
Include requirements assigned to Simulink blocks in the generated code comments (for more information, see “Including Requirements Information with Generated Code” in the Simulink Verification and Validation documentation)	Requirements in block comments.

When you select **Simulink block descriptions**,

- The description text for blocks and Stateflow objects and block names generated as comments can include international (non-US-ASCII) characters. (For details on international character support, see “Support for International (Non-US-ASCII) Characters” in the Real-Time Workshop documentation.)
- The Real-Time Workshop software automatically inserts comments into the generated code for custom blocks. Therefore, it is not necessary to include block comments in the associated TLC file for a custom block.

Note If you have existing TLC files with manually inserted comments for block descriptions, the code generation process emits these comments instead of the automatically generated comments. Consider removing existing block comments from your TLC files. Manually inserted comments might be poorly formatted in the generated code and code-to-model traceability might not work.

- For virtual blocks or blocks that have been removed due to block reduction, no comments are generated.

For more information, see “Real-Time Workshop Pane: Comments” in the Real-Time Workshop reference documentation.

Adding Custom Comments

This procedure allows you to add a comment just above a signal or parameter's identifier in the generated code. This is accomplished using

- A MATLAB or TLC function that you write and save in a `.m` or `.t1c` file
- The **Custom comments (MPT objects only)** check box on the **Comments** pane of the Configuration Parameters dialog box
- Selecting the `.m` or `.t1c` file in the **Custom comments function** field on the **Comments** pane of the Configuration Parameters dialog box.

You may include at least some or all of the property values for the data object. Each Simulink data object (signal or parameter) has properties, as described in Parameter and Signal Property Values on page 6-2. This example comment contains some of the property values for the data object MAP as specified on the Model Explorer:

```
/*      DocUnits:          PSI                               */
/*      Owner:             */                               */
/*      DefinitionFile: specialDef                         */
real_T MAP = 0.0;
```

You can type text in the **Description** field on the Model Explorer for a signal or parameter data object. If you do, and if you select the **Simulink data object descriptions** check box on the **Comments** pane of the Configuration Parameters dialog box, this text will appear beside the signal's or parameter's identifier in the generated code as a comment. This is true whether or not you select the **Custom comments (MPT objects only)** check box discussed in this procedure. For example, typing Manifold Absolute Pressure in the **Description** field for the data object MAP always will result in the following in the generated code:

```
real_T MAP = 0.0;      /* Manifold Absolute Pressure */
```

- 1 Write a MATLAB or TLC function that places comments in the generated files as desired. An example `.m` file named `rtwdemo_comments_mptfun.m` is provided in the `matlab/toolbox/rtw/rtwdemos` directory. This file contains instructions.

The MATLAB function must have three arguments that correspond to `objectName`, `modelName`, and `request`, respectively. The TLC function

must have three arguments that correspond to `objectRecord`, `modelName`, and `request`, respectively. Note also, in the case of the TLC file, you can use the library function `LibGetSLDataObjectInfo` to get every property value of the data object.

- 2** Save the function as a `.m` file or a `.t1c` file with the desired filename and place it in any folder in the MATLAB path.
- 3** Open the model and the Configuration Parameters dialog box.
- 4** Click **Comments** under **Real-Time Workshop** on the left pane. The **Comments** pane appears on the right.
- 5** Select the **Custom comments (MPT objects only)** check box.
- 6** In the **Custom comments function** field, either type the filename of the `.m` file or `.t1c` file you created, or select this filename using the **Browse** button.
- 7** Click the **Apply** button.
- 8** Click **Generate Code**.
- 9** Open the generated files and inspect their content to ensure the comments are what you want.

Adding Global Comments

- “Introduction” on page 16-5
- “Using a Simulink DocBlock to Add a Comment” on page 16-6
- “Using a Simulink Annotation to Add a Comment” on page 16-7
- “Using a Stateflow Note to Add a Comment” on page 16-8
- “Using Sorted Notes to Add Comments” on page 16-9

Introduction

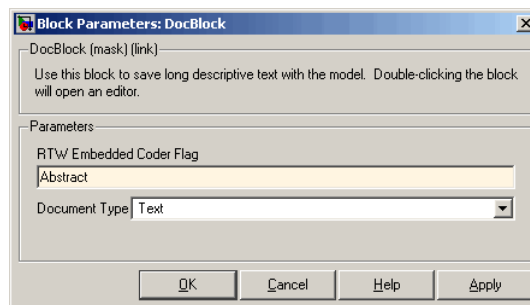
The procedures in this section explain how to add a global comment to a Simulink model so that the comment text appears in the generated file or files where desired. This is accomplished by specifying a template symbol name

with a Simulink DocBlock, a Simulink annotation, or a Stateflow note, or by using a sorted-notes capability that works with Simulink annotations or Stateflow notes (but not DocBlocks). For more information about template symbols, see “Template Symbols and Rules” on page 16-58.

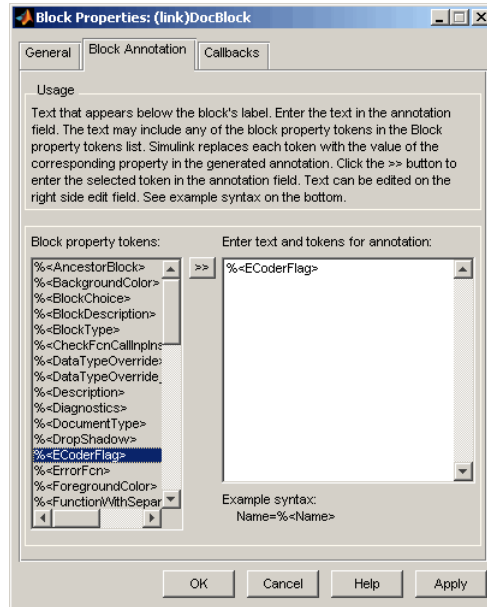
Note Template symbol names `Description` and `ModifiedHistory`, referenced below, also are fields in the Model Properties dialog box. If you use one of these symbol names for global comment text, and its Model Properties field has text in it too, both will appear in the generated files.

Using a Simulink DocBlock to Add a Comment

- 1 With the model open, select **Library Browser** from the **View** menu.
- 2 Drag the DocBlock from **Model-Wide Utilities** in the Simulink library onto the model.
- 3 After double-clicking the DocBlock and typing the desired comment in the editor, save and close the editor. See DocBlock in the Simulink documentation for details.
- 4 Right-click the DocBlock and select **Mask Parameters**. The Block Parameters dialog box appears.
- 5 Type one of the following Documentation child into the **RTW Embedded Coder Flag** field, illustrated below, and then click **OK**: `Abstract`, `Description`, `History`, `ModifiedHistory`, or `Notes`. Template symbol names are case sensitive.



- 6 In the Block Properties dialog box, **Block Annotation** tab, select `%<ECoderFlag>` as shown in the figure below, and then click **OK**. The symbol name typed in the previous step now appears under the DocBlock on the model.



- 7 Save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file.
- 8 To add one or more other comments to the generated files, repeat steps 1 through 7 as desired.

Using a Simulink Annotation to Add a Comment

- 1 Double-click the unoccupied area on the model where you want to place the comment. See “Annotating Diagrams” in the Simulink documentation for details.

Note If you want the code generator to sort multiple comments for the Notes symbol name, replace the next step with “Using Sorted Notes to Add Comments” on page 16-9.

- 2 Type <S:Symbol_name> followed by the comment, where Symbol_name is one of the following Documentation child : Abstract, Description, History, ModifiedHistory, or Notes. For example, type <S:Description>This is the description I want. Template symbol names are case sensitive. (The "S" before the colon indicates "symbol.")
- 3 Click outside the rectangle and save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file.
- 4 To add one or more other comments to the generated files, repeat steps 1 through 3 as desired.

Using a Stateflow Note to Add a Comment

- 1 Right-click the desired unoccupied area on the Stateflow chart where you want to place the comment. See “Using Notes to Extend Charts” in the Stateflow documentation for details.
- 2 Select Add Note from the drop down menu.

Note If you want the code generator to sort multiple comments for the Notes symbol name, replace the next step with “Using Sorted Notes to Add Comments” on page 16-9.

- 3 Type <S:Symbol_name> followed by the comment, where Symbol_name is one of the following Documentation child : Abstract, Description, History, ModifiedHistory, or Notes. For example, type <S:Description>This is the description I want. Template symbol names are case sensitive.

- 4 Click outside the note and save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file.
- 5 To add one or more other comments to the generated files, repeat steps 1 through 4 as desired.

Using Sorted Notes to Add Comments

The sorted-notes capability allows you to add automatically sorted comments to the generated files. The code generator places these comments in each generated file at the location that corresponds to where the `Notes` symbol is located in the template file.

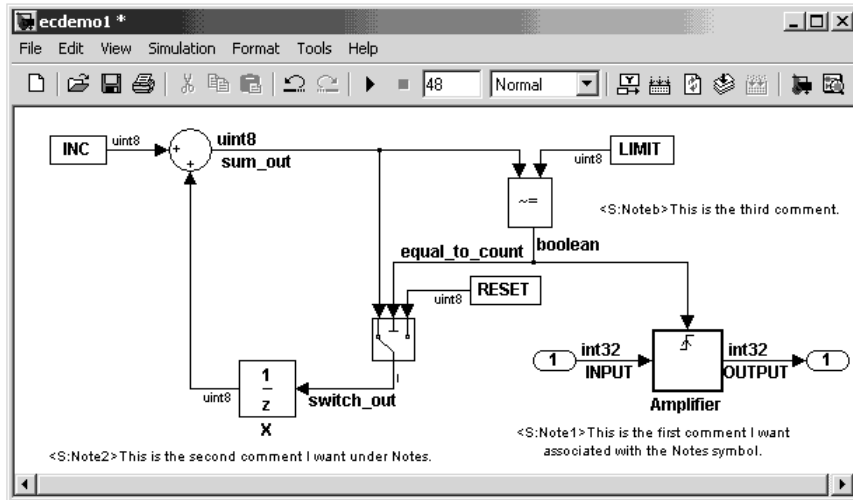
The sorting order the code generator uses is

- Numbers before letters
- Among numbers, 0 is first
- Among letters, uppercase are before lowercase.

You can use sorted notes with a Simulink annotation or a Stateflow note, but not with a DocBlock:

- In the Simulink annotation or the Stateflow note, type `<S:NoteY>` followed by the first comment, where Y is a number or letter.
- Repeat for as many additional comments you want, except replace Y with a subsequent number or letter.

The figure below illustrates sorted notes on a model, and where the code generator places each in a generated file.



Here is the relevant fragment from the generated file for the above model:

```
** NOTES
```

```
** Note1: This is the first comment I want
associated with the Notes symbol.
```

```
Note2: This is the second comment I want under Notes.
```

```
Noteb: This is the third comment.
```

```
**
```

Configuring the Appearance of Generated Identifiers

Customizing Generated Identifiers

Several parameters are available for customizing generated symbols.

To...	Specify...
Define a macro string that specifies whether, and in what order, certain substrings are included within generated identifiers for global variables, global types, field names of global types, subsystem methods, subsystem method arguments, local temporary variables, local block output variables, and constant macros	The macro string with the Identifier format control parameter (for details on how to specify formats, see “Specifying Identifier Formats” on page 16-12 and for limitations, see “Identifier Format Control Parameters Limitations” on page 16-19).
Specify the minimum number of characters the code generator uses for mangled symbols	Specify an integer value for the Minimum mangle length (for details, see “Name Mangling” on page 16-15).
Specify the maximum number of characters the code generator can use for function, typedef, and variable names (default 31)	Specify an integer value for the Maximum identifier length . If you expect your model to generate lengthy identifiers (due to use of long signal or parameter names, for example), or you find that identifiers are being mangled more than expected, you should increase the value of this parameter.
Control whether scalar inlined parameter values are expressed in generated code as literal values or macros	The value Literals or Macros for the Generate scalar inlined parameters as parameter <ul style="list-style-type: none"> • Literals: Parameters are expressed as numeric constants and takes effect if Inline parameters is selected. • Macros: Parameters are expressed as variables (with <code>#define</code> macros). This setting makes code more readable.

For more information, see “Real-Time Workshop Pane: Symbols” in the Real-Time Workshop reference documentation.

Configuring Symbols

- “Specifying Simulink Data Object Naming Rules” on page 16-12
- “Specifying Identifier Formats” on page 16-12
- “Name Mangling” on page 16-15
- “Traceability” on page 16-16
- “Minimizing Name Mangling” on page 16-17
- “Model Referencing Considerations” on page 16-18
- “Exceptions to Identifier Formatting Conventions” on page 16-18
- “Identifier Format Control Parameters Limitations” on page 16-19

Specifying Simulink Data Object Naming Rules

To Define Rules that Change the Names of a Model’s...	Specify a Naming Rule with the ...
Signals	Signal naming parameter
Parameters	Parameter naming parameter
Parameters that have a storage class of Define	#define naming parameter

For more information on these parameters, see “Specifying Simulink Data Object Naming Rules” on page 11-34 in the Real-Time Workshop Embedded Coder Module Packaging Features document.

Specifying Identifier Formats

The **Identifier format control** parameters let you customize generated identifiers by entering a macro string that specifies whether, and in what order, certain substrings are included within generated identifiers. For example, you can specify that the root model name be inserted into each identifier.

The macro string can include

- Tokens of the form \$X, where X is a single character. Valid tokens are listed in Identifier Format Tokens on page 16-13. You can use or omit tokens as you want, with the exception of the \$M token, which is required (see “Name Mangling” on page 16-15) and subject to the use and ordering restrictions noted in Identifier Format Control Parameter Values on page 16-14.
- Any valid C or C++ language identifier characters (a-z, A-Z, _ , 0-9).

The build process generates each identifier by expanding tokens (in the order listed in Identifier Format Tokens on page 16-13) and inserting the resultant strings into the identifier. Character strings between tokens are simply inserted directly into the identifier. Contiguous token expansions are separated by the underscore (_) character.

Identifier Format Tokens

Token	Description
\$M	Insert name mangling string if required to avoid naming collisions (see “Name Mangling” on page 16-15). Note: This token is required.
\$F	Insert method name (for example, _Update for update method). This token is available only for subsystem methods.
\$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
\$R	Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Note that when using model referencing, this token is required in addition to \$M (see “Model Referencing Considerations” on page 16-18). Note: This token replaces the Prefix model name to global identifiers option used in previous releases.

Identifier Format Tokens (Continued)

Token	Description
\$H	<p>Insert tag indicating system hierarchy level. For root-level blocks, the tag is the string <code>root_</code>. For blocks at the subsystem level, the tag is of the form <code>SN_</code>, where N is a unique system number assigned by the Simulink software. This token is available only for subsystem methods and field names of global types.</p> <p>Note: This token replaces the Include System Hierarchy Number in Identifiers option used in previous releases.</p>
\$A	<p>Insert data type acronym (for example, <code>i32</code> for long integers) to signal and work vector identifiers. This token is available only for local block output variables and field names of global types.</p> <p>Note: This token replaces the Include data type acronym in identifier option used in previous releases.</p>
\$I	<p>Insert <code>u</code> if the argument is an input or <code>y</code> if the argument is an output, (for example, <code>rtu_</code> for an input argument and <code>rty_</code> for an output argument). This token is available only for subsystem method arguments.</p>

Identifier Format Control Parameter Values on page 16-14 lists the default macro string, the supported tokens, and the applicable restrictions for each **Identifier format control** parameter.

Identifier Format Control Parameter Values

Parameter	Default Value	Supported Tokens	Restrictions
Global variables	\$R\$N\$M	\$R, \$N, \$M	\$F, \$H, \$A, and \$I are disallowed.
Global types	\$N\$R\$M	\$N, \$R, \$M	\$F, \$H, \$A, and \$I are disallowed.
Field name of global types	\$N\$M	\$N, \$M, \$H, \$A	\$R, \$F, and \$I are disallowed.

Identifier Format Control Parameter Values (Continued)

Parameter	Default Value	Supported Tokens	Restrictions
Subsystem methods	\$R\$N\$M\$F	\$R, \$N, \$M, \$F, \$H	\$F and \$H are empty for Stateflow functions; \$A and \$I are disallowed.
Subsystem method arguments	rtu_{\$N\$M} or rty_{\$N\$M}	\$N, \$M, \$I	\$R, \$F, \$H, and \$A are disallowed.
Local temporary variables	{\$N\$M}	\$N, \$M, \$R	\$F, \$H, \$A, and \$I are disallowed.
Local block output variables	rtb_{\$N\$M}	\$N, \$M, \$A	\$R, \$F, \$H, and \$I are disallowed.
Constant macros	\$R\$N\$M	\$R, \$N, \$M	\$F, \$H, \$A, and \$I are disallowed.

Non-ERT based targets (such as the GRT target) implicitly use a default \$R\$N\$M specification. This specifies identifiers consisting of the root model name, followed by the name of the generating object (signal, parameter, state, and so on), followed by a name mangling string (see “Name Mangling” on page 16-15).

For limitations that apply to **Identifier format control** parameters, see “Identifier Format Control Parameters Limitations” on page 16-19.

Name Mangling

In identifier generation, a circumstance that would cause generation of two or more identical identifiers is called a *name collision*. Name collisions are never permissible. When a potential name collision exists, unique *name mangling* strings are generated and inserted into each of the potentially conflicting identifiers. Each name mangling string is guaranteed to be unique for each generated identifier.

The position of the **\$M** token in the **Identifier format control** parameter specification determines the position of the name mangling string in the generated identifiers. For example, if the specification **\$R\$N\$M** is used, the name mangling string is appended (if required) to the end of the identifier.

The **Minimum mangle length** parameter specifies the minimum number of characters used when a name mangling string is generated. The default is 1 character. As described below, the actual length of the generated string may be longer than this minimum.

Traceability

An important aspect of model based design is the ability to generate identifiers that can easily be traced back to the corresponding entities within the model. To ensure traceability, it is important to make sure that incremental revisions to a model have minimal impact on the identifier names that appear in generated code. There are two ways of achieving this:

- 1 Choose unique names for Simulink objects (blocks, signals, states, and so on) as much as possible.
- 2 Make use of name mangling when conflicts cannot be avoided.

When conflicts cannot be avoided (as may be the case in models that use libraries or model reference), name mangling ensures traceability. The position of the name mangling string is specified by the placement of the **\$M** token in the **Identifier format control** parameter specification. Mangle characters consist of lower case characters (a-z) and numerics (0-9), which are chosen with a checksum that is unique to each object. How Name Mangling Strings Are Computed on page 16-16 describes how this checksum is computed for different types of objects.

How Name Mangling Strings Are Computed

Object Type	Source of Mangling String
Block diagram	Name of block diagram
Simulink block	Full path name of block

How Name Mangling Strings Are Computed (Continued)

Object Type	Source of Mangling String
Simulink parameter	Full name of parameter owner (that is, model or block) and parameter name
Simulink signal	Signal name, full name of source block, and port number
Stateflow objects	Complete path to Stateflow block and Stateflow computed name (unique within chart)

The length of the name mangling string is specified by the **Minimum mangle length** parameter. The default value is 1, but this automatically increases during code generation as a function of the number of collisions.

To minimize disturbance to the generated code during development, specify a larger **Minimum mangle length**. A **Minimum mangle length** of 4 is a conservative and safe value. A value of 4 allows for over 1.5 million collisions for a particular identifier before the mangle length is increased.

Minimizing Name Mangling

Note that the length of generated identifiers is limited by the **Maximum identifier length** parameter. When a name collision exists, the \$M token is always expanded to the minimum number of characters required to avoid the collision. Other tokens and character strings are expanded in the order listed in Identifier Format Tokens on page 16-13. If the **Maximum identifier length** is not large enough to accommodate full expansions of the other tokens, partial expansions are used. 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.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate.

Set the **Minimum mangle length** parameter to reserve at least three characters for the name mangling string. The length of the name mangling string increases as the number of name collisions increases.

Note that an existing name mangling string increases (or decreases) in length if changes to model create more (or fewer) collisions. If the length of the name mangling string increases, additional characters are appended to the existing string. For example, 'xyz' might change to 'xyzQ'. In the inverse case (fewer collisions) 'xyz' would change to 'xy'.

Model Referencing Considerations

Within a model that uses model referencing, there can be no collisions between the names of the constituent models. When generating code 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. A code generation error occurs if **Maximum identifier length** is not large enough.

When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the identifier from the referenced model is preserved. Name mangling is performed on the identifier from the higher-level model.

Exceptions to Identifier Formatting Conventions

There are some exceptions to the identifier formatting conventions described above:

- Type name generation: The above name mangling conventions do not apply to type names (that is, **typedef** statements) generated for global data types. If the **\$R** token is included in the **Identifier format control** parameter specification, the model name is included in the **typedef**. The **Maximum identifier length** parameter is not respected when generating type definitions.
- Non-Auto storage classes: The **Identifier format control** parameter specification does not affect objects (such as signals and parameters)

that have a storage class other than Auto (such as ImportedExtern or ExportedGlobal).

Identifier Format Control Parameters Limitations

The following limitations apply to the **Identifier format control** parameters:

- The following autogenerated identifiers currently do not fully comply with the setting of the **Maximum identifier length** parameter on the **Real-Time Workshop/Symbols** pane of the Configuration Parameters dialog box.
 - Model methods
 - The applicable format string is `RF`, and the longest `$F` is `_derivatives`, which is 12 characters long. The model name can be up to 19 characters without exceeding the default **Maximum identifier length** of 31.
 - Local functions generated by S-functions or by add-on products such as Signal Processing Blockset™ that rely on S-functions
 - Local variables generated by S-functions or by add-on products such as Signal Processing Blockset that rely on S-functions
 - DWork identifiers generated by S-functions in referenced models
 - Fixed-point shared utility macros or shared utility functions
 - Simulink rtm macros
 - Most are within the default **Maximum identifier length** of 31, but some exceed the limit. Examples are `RTMSpecAccsGetStopRequestedValStoredAsPtr`, `RTMSpecAccsGetErrorStatusPointer`, and `RTMSpecAccsGetErrorStatusPointerPointer`.
 - Define protection guard macros
 - Header file guards, such as `_RTW_HEADER_$(filename)_h_`, which can exceed the default **Maximum identifier length** of 31 given a filename such as `$R_private.h`.
 - Include file guards, such as `_$R_COMMON_INCLUDES_`.
 - typedef guards, such as `_CSCI_$R_CHARTSTRUCT_`.

- In some situations, the following identifiers potentially can conflict with others.
 - Model methods
 - Local functions generated by S-functions or by add-on products such as Signal Processing Blockset that rely on S-functions
 - Local variables generated by S-functions or by add-on products such as Signal Processing Blockset that rely on S-functions
 - Fixed-point shared utility macros or shared utility functions
 - Include header guard macros
- The following external identifiers that are unknown to the Simulink software might conflict with autogenerated identifiers.
 - Identifiers defined in custom code
 - Identifiers defined in custom header files
 - Identifiers introduced through a non-ANSI C standard library
 - Identifiers defined by custom TLC code
- Identifiers generated for simulation targets may exceed the **Maximum identifier length**. Simulation targets include the model reference simulation target, the accelerated simulation target, the RSim target, and the S-function target.

Controlling Code Style

You can control the following style aspects in generated code:

- Level of parenthesization
- Whether to preserve order of operands in expressions
- Whether to preserve empty primary condition expressions in `if` statements
- Whether to generate code for `if-elseif-else` decision logic as `switch-case` statements
- Whether to include the `extern` keyword in function declarations

For example, C code contains some syntactically required parentheses, and can contain additional parentheses that change semantics by overriding default operator precedence. C code can also contain optional parentheses that have no functional significance, but serve only to increase the readability of the code. Optional C parentheses vary between two stylistic extremes:

- Include the minimum parentheses required by C syntax and any precedence overrides, so that C precedence rules specify all semantics unless overridden by parentheses.
- Include the maximum parentheses that can exist without duplication, so that C precedence rules become irrelevant: parentheses alone completely specify all semantics.

Understanding code with minimum parentheses can require correctly applying nonobvious precedence rules, but maximum parentheses can hinder code reading by belaboring obvious precedence rules. Various parenthesization standards exist that specify one or the other extreme, or define an intermediate style that can be useful to human code readers.

You control the code style options by setting parameters on the **Real-Time Workshop > Code Style** pane. For details on the parameters, see “Real-Time Workshop Pane: Code Style” in the Real-Time Workshop Embedded Coder reference documentation.

Configuring Templates for Customizing Code Organization and Format

In this section...

“Overview” on page 16-23

“Custom File Processing Components” on page 16-24

“Custom File Processing User Interface Options” on page 16-24

“Code Generation Template (CGT) Files” on page 16-26

“Using Custom File Processing (CFP) Templates” on page 16-30

“Custom File Processing (CFP) Template Structure” on page 16-30

“Changing the Organization of a Generated File” on page 16-32

“Generating Source and Header Files with a Custom File Processing (CFP) Template” on page 16-34

“Comparison of a Template and Its Generated File” on page 16-43

“Code Template API Summary” on page 16-46

“Generating Custom File and Function Banners” on page 16-49

“Template Symbols and Rules” on page 16-58

Customize generated code using code and data templates

To...	Enter or Select...
Specify a template that defines the top-level organization and formatting of generated source code (.c or .cpp) files	Enter a code generation template (CGT) file for the Source file (*.c) template parameter.
Specify a template that defines the top-level organization and formatting of generated header (.h) files	Enter a CGT file for the Header file (*.h) template parameter. This template file can be the same template file that you specify for Source file (.c) template . If you use the same template file, source and header files contain identical banners. The default template is <code>matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt</code> .

To...	Enter or Select...
Specify a template that organizes generated code into sections (such as includes, typedefs, functions, and more)	Enter a custom file processing (CFP) template file for the “File customization template” parameter. A CFP template can emit code, directives, or comments into each section. For more information, see “Using Custom File Processing (CFP) Templates” on page 16-30.
Generate a model-specific example main program module	Select Generate an example main program . For more information, see “Generating a Standalone Program” on page 31-3.

Note Place the template files that you specify on the MATLAB path.

Overview

This section describes Real-Time Workshop Embedded Coder *custom file processing* (CFP) features. Custom file processing simplifies generation of custom source code. You can:

- Generate any type of source (.c or .cpp) or header (.h) file. Using a *custom file processing template* (CFP template), you can control how code emits to the standard generated model files (for example, *model.c* or *model.cpp*, *model.h*) or generate files that are independent of model code.
- Organize generated code into sections (such as includes, typedefs, functions, and more). Your CFP template can emit code (for example, functions), directives (such as `#define` or `#include` statements), or comments into each section.
- Generate custom *file banners* (comment sections) at the start and end of generated code files and custom *function banners* that precede functions in the generated code.
- Generate code to call model functions, such as *model_initialize*, *model_step*, and so on.
- Generate code to read and write model inputs and outputs.
- Generate a main program module.
- Obtain information about the model and the generated files from the model.

Custom File Processing Components

The custom file processing features are based on the following interrelated components:

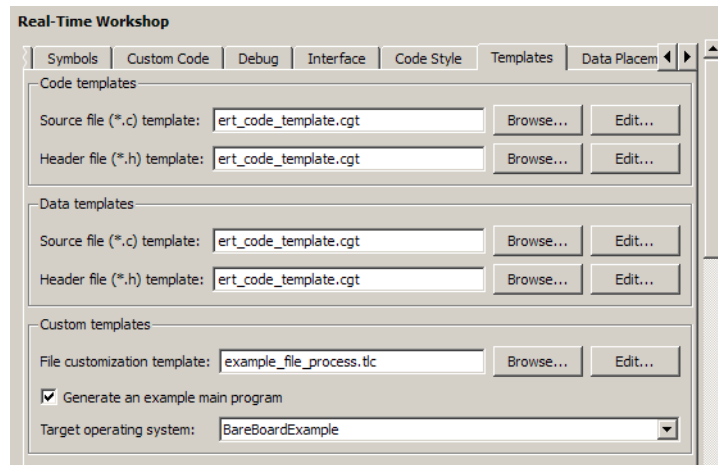
- *Code generation template (CGT) files*: a CGT file defines the top-level organization and formatting of generated code. See “Code Generation Template (CGT) Files” on page 16-26.
- *The code template API*: a high-level Target Language Compiler (TLC) API that provides functions with which you can organize code into named sections and subsections of generated source and header files. The code template API also provides utilities that return information about generated files, generate standard model calls, and perform other functions. See “Code Template API Summary” on page 16-46.
- *Custom file processing (CFP) templates*: a CFP template is a TLC file that manages the process of custom code generation. A CFP template assembles code to be generated into buffers. A CFP template also calls the code template API to emit the buffered code into specified sections of generated source and header files. A CFP template interacts with a CGT file, which defines the ordering of major sections of the generated code. See “Using Custom File Processing (CFP) Templates” on page 16-30.

To use CFP templates, you must understand TLC programming. See the Target Language Compiler document.

Custom File Processing User Interface Options

To use custom file processing features, create CGT files and CFP templates. These files are based on default templates provided by the Real-Time Workshop Embedded Coder software. Once you have created your templates, you must integrate them into the code generation process.

Select and edit CGT files and CFP templates, and specify their use in the code generation process in the **Real-Time Workshop > Templates** pane of a model configuration set. The following figure shows all options configured for their defaults.



Real-Time Workshop: Templates Pane

The options related to custom file processing are:

- The **Source file (.c) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating source (.c or .cpp) files. You must place this file on the MATLAB path.
- The **Header file (.h) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating header (.h) files. You must place this file on the MATLAB path.

By default, the template for both source and header files is `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.

- The **File customization template** edit field in the **Custom templates** section. This field specifies the name of a CFP template file to use when generating code files. You must place this file on the MATLAB path. The default CFP template is `matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc`.

In each of these fields, click **Browse** to navigate to and select an existing CFP template or CGT file. Click **Edit** to open the specified file into the MATLAB editor where you can customize it. You must place this file on the

Code Generation Template (CGT) Files

CGT files have the following applications:

- Generation of custom banners (comments sections) in code files. See “Generating Custom File and Function Banners” on page 16-49.
- Advanced features, as described in Defining Data Representation and Storage for Code Generation on page 1 that use CGT files.
- Generation of custom code using a CFP template requires a CGT file. To correctly use CFP templates, you must understand the CGT file structure. In many cases, however, you can use the default CGT file without modifying it.

Default CGT file

The Real-Time Workshop Embedded Coder software provides a default CGT file, `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`. Base your custom CGT files on the default file.

CGT File Structure

A CGT file consists of one required section and four optional sections:

Code Insertion Section. (Required) This section contains tokens that define an ordered partitioning of the generated code into a number of sections (such as `Includes` and `Defines` sections). Tokens have the form of:

```
%<SectionName>
```

For example,

```
%<Includes>
```

The Real-Time Workshop Embedded Coder software defines a minimal set of required tokens. These tokens generate C or C++ source or header code. They are *built-in* tokens (see “Built-In Tokens and Sections” on page 16-27). You can also define *custom* tokens and add them to the code insertion section (see “Generating a Custom Section” on page 16-41).

Each token functions as a placeholder for a corresponding section of generated code. The ordering of the tokens defines the order in which the corresponding

sections appear in the generated code. A token in the CGT file does not guarantee that the corresponding section is generated. To generate code into a given section, explicitly call the code template API from a CFP template, as described in “Using Custom File Processing (CFP) Templates” on page 16-30.

The CGT tokens define the high-level organization of generated code. Using the code template API, you can partition each code section into named subsections, as described in “Subsections” on page 16-29.

In the code insertion section, you can also insert C or C++ comments between tokens. Such comments emit directly into the generated code.

File Banner Section. (Optional) This section contains comments and tokens you use in generating a custom file banner. See “Generating Custom File and Function Banners” on page 16-49.

Function Banner Section. (Optional) This section contains comments and tokens for use in generating a custom function banner. See “Generating Custom File and Function Banners” on page 16-49.

Shared Utility Function Banner Section. (Optional) This section contains comments and tokens for use in generating a custom shared utility function banner. See “Generating Custom File and Function Banners” on page 16-49.

File Trailer Section. (Optional) This section contains comments for use in generating a custom trailer banner. See “Generating Custom File and Function Banners” on page 16-49.

Built-In Tokens and Sections

The following code extract shows the required code insertion section of the default CGT file with the required built-in tokens.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Code insertion section (required)
%%   These are required tokens. You can insert comments and other tokens in
%%   between them, but do not change their order or remove them.
%%
%<Includes>
%<Defines>
```

```
%<Types>  
%<Enums>  
%<Definitions>  
%<Declarations>  
%<Functions>
```

Note the following requirements for customizing a CGT file:

- Do not remove required built-in tokens.
- Built-in tokens must appear in the order shown because each successive section has dependencies on previous sections.
- Only one token per line.
- Do not repeat tokens.
- You can add custom tokens and comments to the code insertion section as long as you do not violate the previous requirements.

The following table summarizes the built-in tokens and corresponding section names, and describes the code sections.

Built-In CGT Tokens and Corresponding Code Sections

Token and Section Name	Description
Includes	#include directives section
Defines	#define directives section
Types	typedef section. Typedefs can depend on any previously defined type
Enums	Enumerated types section
Definitions	Data definitions (for example, <code>double x = 3.0;</code>)
Declarations	Data declarations (for example, <code>extern double x;</code>)
Functions	C or C++ functions

Subsections

You can define one or more named subsections for any section. Some of the built-in sections have predefined subsections summarized in Subsections Defined for Built-In Sections on page 16-29.

Note Sections and subsections emit to the source or header file in the order listed in the CGT file.

Using the custom section feature, you can define additional sections. See “Generating a Custom Section” on page 16-41.

Subsections Defined for Built-In Sections

Section	Subsections	Subsection Description
Includes	N/A	
Defines	N/A	
Types	IntrinsicTypes	Intrinsic typedef section. Intrinsic types depend only on intrinsic C or C++ types.
Types	PrimitiveTypedefs	Primitive typedef section. Primitive typedefs depend only on intrinsic C or C++ types and on any typedefs previously defined in the IntrinsicTypes section.
Types	UserTop	You can place any type of code in this section, including code that has dependencies on the previous sections.
Types	Typedefs	typedef section. Typedefs can depend on any previously defined type
Enums	N/A	
Definitions	N/A	
Declarations	N/A	
Functions		C or C++ functions
Functions	CompilerErrors	#warning directives

Subsections Defined for Built-In Sections (Continued)

Section	Subsections	Subsection Description
Functions	CompilerWarnings	#error directives
Functions	Documentation	Documentation (comment) section
Functions	UserBottom	You can place any code in this section.

Using Custom File Processing (CFP) Templates

The files provided to support custom file processing are

- *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*: A TLC function library that implements the code template API. *codetemplatelib.tlc* also provides the comprehensive documentation of the API in the comments headers preceding each function.
- *matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc*: An example custom file processing (CFP) template, which you should use as the starting point for creating your own CFP templates. Guidelines and examples for creating a CFP template are provided in “Generating Source and Header Files with a Custom File Processing (CFP) Template” on page 16-34.
- TLC files supporting generation of single-rate and multirate main program modules (see “Customizing Main Program Module Generation” on page 16-39).

Once you have created a CFP template, you must integrate it into the code generation process, using the **File customization template** edit field (see “Custom File Processing User Interface Options” on page 16-24).

Custom File Processing (CFP) Template Structure

A custom file processing (CFP) template imposes a simple structure on the code generation process. The template, a code generation template (CGT) file, partitions the code generated for each file into a number of sections. These sections are summarized in Built-In CGT Tokens and Corresponding Code Sections on page 16-28 and Subsections Defined for Built-In Sections on page 16-29.

Code for each section is assembled in buffers and then emitted, in the order listed, to the file being generated.

To generate a file section, your CFP template must first assemble the code to be generated into a buffer. Then, to emit the section, your template calls the TLC function

```
LibSetSourceFileSection(fileH, section, tmpBuf)
```

where

- `fileH` is a file reference to a file being generated.
- `section` is the code section or subsection to which code is to be emitted. `section` must be one of the section or subsection names listed in Subsections Defined for Built-In Sections on page 16-29.

Determine the `section` argument as follows:

- If Subsections Defined for Built-In Sections on page 16-29 defines no subsections for a given section, use the section name as the `section` argument.
 - If Subsections Defined for Built-In Sections on page 16-29 defines one or more subsections for a given section, you can use either the section name or a subsection name as the `section` argument.
 - If you have defined a custom token denoting a custom section, do not call `LibSetSourceFileSection`. Special API calls are provided for custom sections (see “Generating a Custom Section” on page 16-41).
- `tmpBuf` is the buffer containing the code to be emitted.

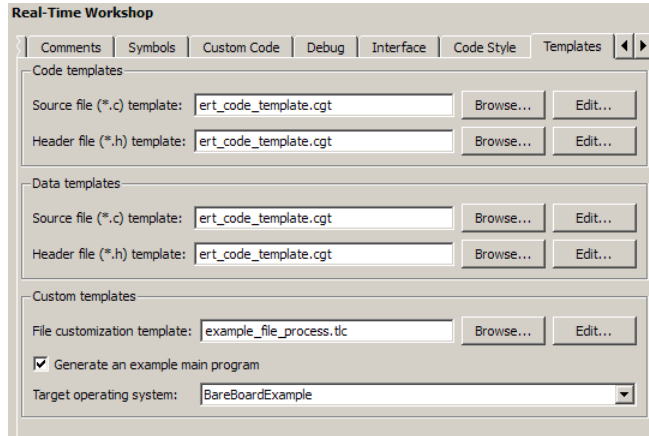
There is no requirement to generate all of the available sections. Your template need only generate the sections you require in a particular file.

Note that no legality or syntax checking is performed on the custom code within each section.

The next section, “Generating Source and Header Files with a Custom File Processing (CFP) Template” on page 16-34, provides typical usage examples.

Changing the Organization of a Generated File

The files you generated in the previous procedures are organized according to the general Real-Time Workshop Embedded Coder template. This template has the filename `ert_code_template.cgt`, and is specified by default in **Templates** pane of the Configuration Parameters dialog box.



The following fragment shows the `rtwdemo_mpf.c` file header that is generated using this default template:

```
/*
 * File: rtwdemo_mpf.c
 *
 * Real-Time Workshop code generated for Simulink model rtwdemo_mpf.
 *
 * Model version                : 1.87
 * Real-Time Workshop file version : 7.3 (R2009a) 20-Nov-2008
 * Real-Time Workshop file generated on : Wed Dec 17 16:27:14 2008
 * TLC version                  : 7.3 (Nov 26 2008)
 * C/C++ source code generated on   : Wed Dec 17 16:27:14 2008
 */
```

You can change the organization of generated files using code templates and data templates. Code templates organize the files that contain functions, primarily. Data templates organize the files that contain identifiers. In this procedure, you organize the generated files, using the supplied MPF code and data templates:

- 1** Display the active Real-Time Workshop **Templates** configuration parameters.
- 2** In the **Code templates** section of the **Templates** pane, type `code_c_template.cgt` into the **Source file (*.c) templates** text box.
- 3** Type `code_h_template.cgt` into the **Header file (*.h) templates** text box.
- 4** In the **Data templates** section, type `data_c_template.cgt` into the **Source file (*.c) templates** text box.
- 5** Type `data_h_template.cgt` into the **Header file (*.h) templates** text box, and click **Apply**.

- 6 Click **Generate code**. Now the files are organized using the templates you specified. For example, the `rtwdemo_mpf.c` file header now is organized like this:

```

/**
*****
** FILE INFORMATION:
** Filename:          rtwdemo_mpf.c
** File Creation Date: 17-Dec-2008
**
** ABSTRACT:
**
**
** NOTES:
**
**
** MODEL INFORMATION:
** Model Name:        rtwdemo_mpf
** Model Description: Data packaging examples
** Model Version:     1.87
** Model Author:      The MathWorks Inc. - Mon Mar 01 11:23:00 2004
**
** MODIFICATION HISTORY:
** Model at Code Generation: username - Wed Dec 17 16:47:16 2008
**
** Last Saved Modification: username - Wed Dec 17 15:23:20 2008
**
*****
**/

```

Generating Source and Header Files with a Custom File Processing (CFP) Template

This section walks you through the process of generating a simple source (.c or .cpp) and header (.h) file using the example CFP template. Then, it examines the template and the code generated by the template.

The example CFP template, `matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc`, demonstrates some of the capabilities of the code template API, including

- Generation of simple source (.c or .cpp) and header (.h) files
- Use of buffers to generate file sections for includes, functions, and so on
- Generation of includes, defines, into the standard generated files (for example, *model.h*)
- Generation of a main program module

Generating Code with a CFP Template

This section sets up a CFP template and configures a model to use the template in code generation. The template generates (in addition to the standard model files) a source file (*timestwo.c* or *.cpp*) and a header file (*timestwo.h*).

Follow the steps below to become acquainted with the use of CFP templates:

- 1** Copy the example CFP template, *matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc*, to a directory outside the MATLAB directory structure (that is, not under *matlabroot*). If the directory is not on the MATLAB path or the TLC path, then add it to the MATLAB path. It is good practice to locate the CFP template in the same directory as your system target file, which is guaranteed to be on the TLC path.
- 2** Rename the copied *example_file_process.tlc* to *test_example_file_process.tlc*.
- 3** Open *test_example_file_process.tlc* into the MATLAB editor.
- 4** Uncomment the following line:

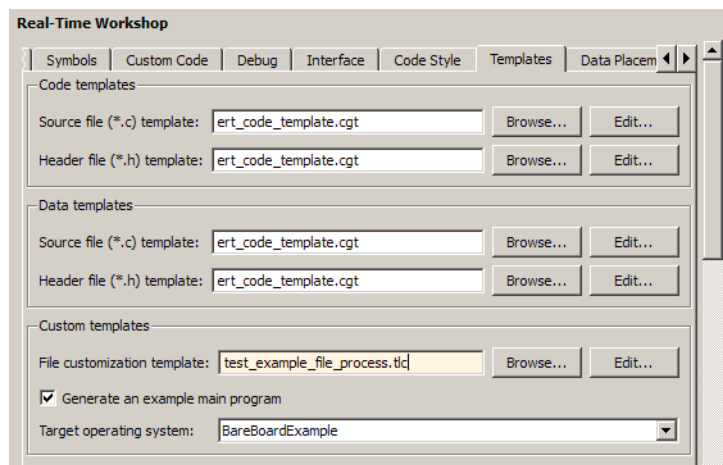
```
%% %assign ERTCustomFileTest = TLC_TRUE
```

It now reads:

```
%assign ERTCustomFileTest = TLC_TRUE
```

If *ERTCustomFileTest* is not assigned as shown, the CFP template is ignored in code generation.

- 5 Save your changes to the file. Keep `test_example_file_process.tlc` open, so you can refer to it later.
- 6 Open the `rtwdemo_udt` model.
- 7 Open the Simulink Model Explorer. Select the active configuration set of the model, and open the **Real-Time Workshop** pane of the active configuration set.
- 8 Click the **Templates** tab.
- 9 Configure the **File customization template** field as shown below. The `test_example_file_process.tlc` file, which you previously edited, is now the specified CFP template for your model.



- 10 Select the **Generate code only** option.
- 11 Click **Apply**.
- 12 Click **Generate code**. During code generation, notice the following message on the MATLAB command window:

```
Warning: Overriding example ert_main.c!
```


This message is displayed because `test_example_file_process.tlc` generates the main program module, overriding the default action of the ERT target. This is explained in greater detail below.

- 13 The `rtwdemo_udt` model is configured to generate an HTML code generation report. After code generation completes, view the report. Notice that the **Generated Files** list contains the files `timestwo.c`, `timestwo.h`, and `ert_main.c`. These files were generated by the CFP template. The next section examines the template to learn how this was done.
- 14 Keep the model, the code generation report, and the `test_example_file_process.tlc` file open so you can refer to them in the next section.

Analysis of the Example CFP Template and Generated Code

This section examines excerpts from `test_example_file_process.tlc` and some of the code it generates. Refer to the comments in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc` while reading the following discussion.

Generating Code Files. Source (`.c` or `.cpp`) and header (`.h`) files are created by calling `LibCreateSourceFile`, as in the following excerpts:

```
%assign cFile = LibCreateSourceFile("Source", "Custom", "timestwo")
...
%assign hFile = LibCreateSourceFile("Header", "Custom", "timestwo")
```

Subsequent code refers to the files by the file reference returned from `LibCreateSourceFile`.

File Sections and Buffers. The code template API lets you partition the code generated to each file into sections, tagged as `Definitions`, `Includes`, `Functions`, `Banner`, and so on. You can append code to each section as many times as required. This technique gives you a great deal of flexibility in the formatting of your custom code files.

Subsections Defined for Built-In Sections on page 16-29 describes the available file sections and their order in the generated file.

For each section of a generated file, use `%openfile` and `%closefile` to store the text for that section in temporary buffers. Then, to write (append) the buffer contents to a file section, call `LibSetSourceFileSection`, passing in the desired section tag and file reference. For example, the following code uses two buffers (`tmwtypesBuf` and `tmpBuf`) to generate two sections (tagged "Includes" and "Functions") of the source file `timestwo.c` or `.cpp` (referenced as `cFile`):

```
%openfile tmwtypesBuf

#include "tmwtypes.h"

%closefile tmwtypesBuf

%<LibSetSourceFileSection(cFile,"Includes",tmwtypesBuf)>

%openfile tmpBuf

/* Times two function */
real_T timestwofcn(real_T input) {
    return (input * 2.0);
}

%closefile tmpBuf

%<LibSetSourceFileSection(cFile,"Functions",tmpBuf)>
```

These two sections generate the entire `timestwo.c` or `.cpp` file:

```
#include "tmwtypes.h"

/* Times two function */
FLOAT64 timestwofcn(FLOAT64 input)
{
    return (input * 2.0);
}
```

Adding Code to Standard Generated Files. The `timestwo.c` or `.cpp` file generated in the previous example was independent of the standard code files generated from a model (for example, `model.c` or `.cpp`, `model.h`, and so on). You can use similar techniques to generate custom code within the model files. The code template API includes functions to obtain the names of the standard models files and other model-related information. The following excerpt calls `LibGetMdlPubHdrBaseName` to obtain the correct name for the `model.h` file. It then obtains a file reference and generates a definition in the `Defines` section of `model.h`:

```
%% Add a #define to the model's public header file model.h

%assign pubName = LibGetMdlPubHdrBaseName()
%assign modelH = LibCreateSourceFile("Header", "Simulink", pubName)

%openfile tmpBuf

#define ACCELERATION 9.81

%closefile tmpBuf

%<LibSetSourceFileSection(modelH, "Defines", tmpBuf)>
```

Examine the generated `rtwdemo_udt.h` file to see the generated `#define` directive.

Customizing Main Program Module Generation. Normally, the ERT target determines whether and how to generate an `ert_main.c` or `.cpp` module based on the settings of the **Generate an example main program** and **Target operating system** options on the **Templates** pane of the Configuration Parameters dialog box. You can use a CFP template to override the normal behavior and generate a main program module customized for your target environment.

To support generation of main program modules, two TLC files are provided:

- `bareboard_srmain.tlc`: TLC code to generate an example single-rate main program module for a bareboard target environment. Code is generated by a single TLC function, `FcnSingleTaskingMain`.

- `bareboard_mrmain.tlc`: TLC code to generate a multirate main program module for a bareboard target environment. Code is generated by a single TLC function, `FcnMultiTaskingMain`.

In the example CFP template file `matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc`, the following code generates either a single- or multitasking `ert_main.c` or `.cpp` module. The logic depends on information obtained from the code template API calls `LibIsSingleRateModel` and `LibIsSingleTasking`:

```
%% Create a simple main. Files are located in MATLAB/rtw/c/tlc/mw.

%if LibIsSingleRateModel() || LibIsSingleTasking()
    %include "bareboard_srmain.tlc"
    %<FcnSingleTaskingMain(>
%else
    %include "bareboard_mrmain.tlc"
    %<FcnMultiTaskingMain(>
%endif
```

Note that `bareboard_srmain.tlc` and `bareboard_mrmain.tlc` use the code template API to generate `ert_main.c` or `.cpp`.

When generating your own main program module, you disable the default generation of `ert_main.c` or `.cpp`. The TLC variable `GenerateSampleERTMain` controls generation of `ert_main.c` or `.cpp`. You can directly force this variable to `TLC_FALSE`. The examples `bareboard_mrmain.tlc` and `bareboard_srmain.tlc` use this technique, as shown in the following excerpt from `bareboard_srmain.tlc`.

```
%if GenerateSampleERTMain
    %assign CompiledModel.GenerateSampleERTMain = TLC_FALSE
    %warning Overriding example ert_main.c!
%endif
```

Alternatively, you can implement a `SelectCallback` function for your target. A `SelectCallback` function is a MATLAB function that is triggered during model loading, and also when the user selects a target with the System Target File browser. Your `SelectCallback` function should deselect and disable the **Generate an example main program** option. This prevents the TLC variable `GenerateSampleERTMain` from being set to `TLC_TRUE`.

See the “rtwgensettings Structure” section in the Real-Time Workshop documentation for information on creating a `SelectCallback` function.

The following code illustrates how to deselect and disable the **Generate an example main program** option in the context of a `SelectCallback` function.

```
slConfigUISetVal(hDlg, hSrc, 'GenerateSampleERTMain', 'off');  
slConfigUISetEnabled(hDlg, hSrc, 'GenerateSampleERTMain', 0);
```

Note Creation of a main program for your target environment requires some customization; for example, in a bareboard environment you need to attach `rt_OneStep` to a timer interrupt. It is expected that you will customize either the generated code, the generating TLC code, or both. See “Guidelines for Modifying the Main Program” on page 31-5 and “Guidelines for Modifying `rt_OneStep`” on page 31-12 for further information.

Generating a Custom Section

You can define custom tokens in a CGT file and direct generated code into an associated built-in section. This feature gives you additional control over the formatting of code within each built-in section. For example, you could add subsections to built-in sections that do not already define any subsections. All custom sections must be associated with one of the built-in sections: `Includes`, `Defines`, `Types`, `Enums`, `Definitions`, `Declarations`, or `Functions`. To create custom sections, you must

- Add a custom token to the code insertion section of your CGT file.
- In your CFP file:
 - Assemble code to be generated to the custom section into a buffer.
 - Declare an association between the custom section and a built-in section, with the code template API function `LibAddSourceFileCustomSection`.
 - Emit code to the custom section with the code template API function `LibSetSourceFileCustomSection`.

The following code examples illustrate the addition of a custom token, `Myincludes`, to a CGT file, and the subsequent association of the custom section `Myincludes` with the built-in section `Includes` in a CFP file.

Note If you have not already created custom CGT and CFP files for your model, copy the default template files *matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt* and *matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc* to a work directory that is outside the MATLAB directory structure but on the MATLAB or TLC path, rename them (for example, add the prefix `test_` to each file), and update the **Templates** pane of the Configuration Parameters dialog box to correctly reference them.

First, add the token `Myincludes` to the code insertion section of your CGT file. For example:

```
%<Includes>
%<Myincludes>
%<Defines>
%<Types>
%<Enums>
%<Definitions>
%<Declarations>
%<Functions>
```

Next, in the CFP file, add code to generate `include` directives into a buffer. For example, in your copy of the example CFP file, you could insert the following section between the `Includes` section and the `Create a simple main` section:

```
% Add a custom section to the model's C file model.c

%openfile tmpBuf
#include "moretables1.h"
#include "moretables2.h"
%closefile tmpBuf

%<LibAddSourceFileCustomSection(modelC, "Includes", "Myincludes")>
%<LibSetSourceFileCustomSection(modelC, "Myincludes", tmpBuf)>
```

The `LibAddSourceFileCustomSection` function call declares an association between the built-in section `Includes` and the custom section `Myincludes`. In effect, `Myincludes` is a subsection of `Includes`.

The `LibSetSourceFileCustomSection` function call directs the code in the `tmpBuf` buffer to the `Myincludes` section of the generated file. `LibSetSourceFileCustomSection` is syntactically identical to `LibSetSourceFileSection`.

In the generated code, the include directives generated to the custom section appear after other code directed to `Includes`.

```
#include "rtwdemo_udt.h"
#include "rtwdemo_udt_private.h"

/* #include "mytables.h" */
#include "moretables1.h"
#include "moretables2.h"
```

Note The placement of the custom token in this example CGT file is arbitrary. By locating `%<Myincludes>` after `%<Includes>`, the CGT file ensures only that the `Myincludes` code appears after `Includes` code.

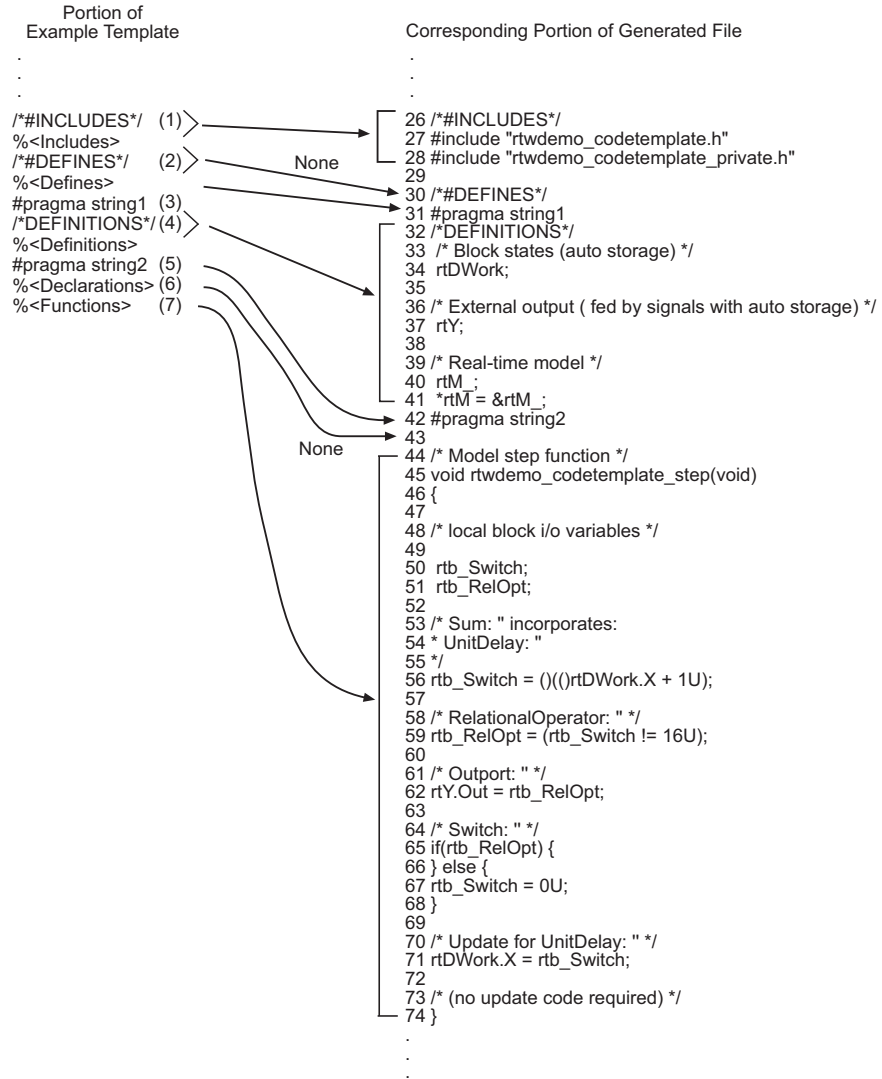
Comparison of a Template and Its Generated File

The next figure shows part of a user-modified MPF template and the resulting Real-Time Workshop Embedded Coder generated code. This figure illustrates how you can use a template to

- Define what code the Real-Time Workshop Embedded Coder software should add to the generated file
- Control the location of code in the file
- Optionally insert comments in the generated file

Notice `%<Includes>`, for example, on the template. The term `Includes` is a symbol name. A percent sign and brackets (`%< >`) must enclose every symbol name. You can add the desired symbol name (within the `%< >` delimiter) at a particular location in the template. This is how you control where the code generator places an item in the generated file.

Template and Generated File



How the Template Affects Code Generation

This part of the template...		Generates in the file...		Explanation
		Line	Description	
(1)	<code>/*#INCLUDES*/ %<Includes></code>	26–28	An <code>/*#INCLUDES*/</code> comment, followed by <code>#include</code> statements	The code generator adds the C/C++ comment as a header, and then interprets the <code>%<Includes></code> template symbol to list all the necessary <code>#include</code> statements in the file. This code is first in this section of the file because the template entries are first.
(2)	<code>/*DEFINES*/ %<Defines></code>	30	A <code>/*DEFINES*/</code> comment, but no <code>#define</code> statements	Next, the code generator places the comment as a header for <code>#define</code> statements, but the file does not need <code>#define</code> . No code is added.
(3)	<code>#pragma string1</code>	31	<code>#pragma</code> statements	While the code generator requires <code>%<></code> delimiters for template symbols, it can also interpret C/C++ statements in the template without delimiters. In this case, the generator adds the specified statements to the code, following the order in which the statements appear in the template.
(5)	<code>#pragma string2</code>	42		
(4)	<code>/*#DEFINITIONS*/ %<Definitions></code>	32–41	<code>/*#DEFINITIONS*/</code> comment, followed by definitions	The code generator places the comment and definitions needed in the file between the <code>#pragma</code> statements, according to the order in the template. It also inserts comments (lines 33 and 36) that are preset in the model's Configuration Parameters dialog box.

How the Template Affects Code Generation (Continued)

This part of the template...		Generates in the file...		Explanation
		Line	Description	
(6)	%<Declarations>	43	No declarations	The file needs no declarations, so the code generator does not generate any for this file. The template has no comment to provide a header. Line 43 is left blank.
(7)	%<Functions>	44–74	Functions	Finally, the code generator adds functions from the model, plus comments that are preset in the Configuration Parameters dialog box. But it adds no comments as a header for the functions, because the template does not have one. This code is last because the template entry is last.

For a list of template symbols and the rules for using them, see “Template Symbol Groups” on page 16-59, “Template Symbols” on page 16-61, and “Rules for Modifying or Creating a Template” on page 16-65. To set comment options, from the **Simulation** menu, select **Configuration Parameters**. On the Configuration Parameters dialog box, select the **Real-Time Workshop > Comments** pane. For details, see “Configuring a Model for Code Generation” in the Real-Time Workshop documentation.

Code Template API Summary

Code Template API Functions on page 16-47 summarizes the code template API. See the source code in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc` for detailed information on the arguments, return values, and operation of these calls.

Code Template API Functions

Function	Description
LibGetNumSourceFiles	Returns the number of created source files (.c or .cpp and .h).
LibGetSourceFileTag	Returns <filename>_h and <filename>_c for header and source files, respectively, where filename is the name of the model file.
LibCreateSourceFile	Creates a new C or C++ file and returns its reference. If the file already exists, simply returns its reference.
LibGetSourceFileFromIdx	Returns a model file reference based on its index. This is useful for a common operation on all files, such as to set the leading file banner of all files.
LibSetSourceFileSection	Adds to the contents of a specified section within a specified file (see also “Custom File Processing (CFP) Template Structure” on page 16-30).
LibIndentSourceFile	Indents a file with the Real-Time Workshop <code>c_indent</code> utility (from within the TLC environment).
LibCallModelInitialize	Returns code for calling the model’s <code>model_initialize</code> function (valid for ERT only).
LibCallModelStep	Returns code for calling the model’s <code>model_step</code> function (valid for ERT only).
LibCallModelTerminate	Returns code for calling the model’s <code>model_terminate</code> function (valid for ERT only).
LibCallSetEventForThisBaseStep	Returns code for calling the model’s <code>set events</code> function (valid for ERT only).

Code Template API Functions (Continued)

Function	Description
LibWriteModelData	Returns data for the model (valid for ERT only).
LibSetRTModelErrorStatus	Returns the code to set the model error status.
LibGetRTModelErrorStatus	Returns the code to get the model error status.
LibIsSingleRateModel	Returns true if model is single rate and false otherwise.
LibGetModelName	Returns name of the model (no extension).
LibGetMdlSrcBaseName	Returns the name of model's main source file (for example, <i>model.c</i> or <i>.cpp</i>).
LibGetMdlPubHdrBaseName	Returns the name of model's public header file (for example, <i>model.h</i>).
LibGetMdlPrvHdrBaseName	Returns the name of the model's private header file (for example, <i>model_private.h</i>).
LibIsSingleTasking	Returns true if the model is configured for single-tasking execution.
LibWriteModelInput	Returns the code to write to a particular root input (that is, a model inport block). (valid for ERT only).
LibWriteModelOutput	Returns the code to write to a particular root output (that is, a model outport block). (valid for ERT only).
LibWriteModelInputs	Returns the code to write to root inputs (that is, all model inport blocks). (valid for ERT only)
LibWriteModelOutputs	Returns the code to write to root outputs (that is, all model outport blocks). (valid for ERT only).

Code Template API Functions (Continued)

Function	Description
<code>LibNumDiscreteSampleTimes</code>	Returns the number of discrete sample times in the model.
<code>LibSetSourceFileCodeTemplate</code>	Set the code template to be used for generating a specified source file.
<code>LibSetSourceFileOutputDirectory</code>	Set the directory into which a specified source file is to be generated.
<code>LibAddSourceFileCustomSection</code>	Add a custom section to a source file. The custom section must be associated with one of the built-in (required) sections: <code>Includes</code> , <code>Defines</code> , <code>Types</code> , <code>Enums</code> , <code>Definitions</code> , <code>Declarations</code> , or <code>Functions</code> .
<code>LibSetSourceFileCustomSection</code>	Adds to the contents of a specified custom section within a specified file. The custom section must have been previously created with <code>LibAddSourceFileCustomSection</code> .
<code>LibGetSourceFileCustomSection</code>	Returns the contents of a specified custom section within a specified file.
<code>LibSetCodeTemplateComplianceLevel</code>	This function must be called from your CFP template before any other code template API functions are called. Pass in 2 as the <code>level</code> argument.

Generating Custom File and Function Banners

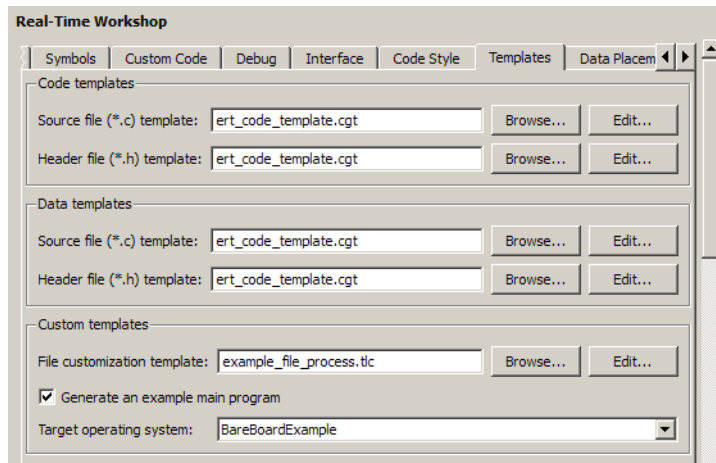
Using code generation template (CGT) files, you can specify custom file banners and function banners for the generated code files. File banners are comment sections in the header and trailer sections of a generated file. Function banners are comment sections for each function in the generated code. Use these banners to add a company copyright statement, specify a special version symbol for your configuration management system, remove time stamps, and for many other purposes. These banners can contain characters, which propagate to the generated code.

To specify banners, create a custom CGT file with customized banner sections. The build process creates an executable TLC file from the CGT file. The code generation process then invokes the TLC file.

You do not need to be familiar with TLC programming to generate custom banners. You can modify example files that are supplied with the ERT target.

Note Prior releases supported direct use of customized TLC files as banner templates. You specified these with the **Source file (.c) banner template** and **Header file (.h) banner template** options of the ERT target. You can still use a custom TLC file banner templates, however, you can now use CGT files instead.

ERT template options on the **Real-Time Workshop > Templates** pane of a configuration set, in the **Code templates** section, support banner generation.



Real-Time Workshop: Templates Pane

The options for function and file banner generation are:

- “Code templates: Source file (*.c) template”: CGT file to use when generating source (.c or .cpp) files. Place this file on the MATLAB path.

- “Code templates: Header file (*.h) template”: CGT file to use when generating header (.h) files. You must place this file on the MATLAB path. This file can be the same template specified in the **Code templates: Source file (*.c) template** field, in which case identical banners are generated in source and header files.

By default, the template for both source and header files is `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.

- In each of these fields, click **Browse** to navigate to and select an existing CGT file for use as a template. Click **Edit** to open the specified file into the MATLAB editor, where you can customize it.

Creating a Custom File and Function Banner Template

To customize a CGT file for custom banner generation, make a local copy of the default code template and edit it, as follows:

- 1** Activate the configuration set you that want to work with.
- 2** Open the **Real-Time Workshop** pane of the active configuration set.
- 3** Click the **Templates** tab.
- 4** By default, the code template specified in the **Code templates: Source file (*.c) template** and **Code templates: Header file (*.h) template** fields is `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
- 5** If you want to use a different template as your starting point, click **Browse** to locate and select a CGT file.
- 6** Click **Edit** button to open the CGT file into the MATLAB editor.
- 7** Save a local copy of the CGT file. Store the copy in a directory that is outside of the MATLAB directory structure, but on the MATLAB path. If necessary, add the directory to the MATLAB path.
- 8** If you intend to use the CGT file with a custom target, locate the CGT file in a folder under your target root directory.
- 9** Rename your local copy of the CGT file. When you rename the CGT file, update the associated **Code templates: Source file (*.c) template** or **Code templates: Header file (*.h) template** field to match the new file name.
- 10** Edit and customize the local copy of the CGT file for banner generation, using the information provided in “Customizing a Code Generation Template (CGT) File for File and Function Banner Generation” on page 16-53.
- 11** Save your changes to the CGT file.
- 12** Click **Apply** to update the configuration set.
- 13** Save your model.

- 14** Generate code. Examine the generated source and header files to confirm that they contain the banners specified by the template or templates.

Customizing a Code Generation Template (CGT) File for File and Function Banner Generation

This section describes how to edit a CGT file for custom file and function banner generation. For a description of CGT files, see “Code Generation Template (CGT) Files” on page 16-26.

Components of the File and Function Banner Sections in the CGT file.

In a CGT file, you can modify the following sections: file banner, function banner, shared utility function banner, and file trailer. Each section is defined by open and close tags. The tags specific to each section are shown in the following table.

CGT File Section	Open Tag	Close Tag
“File Banner” on page 16-55	<FileBanner>	</FileBanner>
“Function Banner” on page 16-56	<FunctionBanner>	</FunctionBanner>
“Shared Utility Function Banner” on page 16-57	<SharedUtilityBanner>	</SharedUtilityBanner>
“File Trailer” on page 16-58	<FileTrailer>	</FileTrailer>

You can customize your banners by including tokens and comments between the open and close tag for each section. Tokens are typically TLC variables, for example <ModelVersion>, which are replaced with values in the generated code.

Note Including C comment indicators, `/*` or `*/`, in the contents of your banner might introduce an error in the generated code.

An open tag includes tag attributes. Enclose the value of the attribute in double quotes. The attributes available for an open tag are:

- width: specifies the width of the file or function banner comments in the generated code. The default value is 80.
- style: specifies the boundary for the file or function banner comments in the generated code. See Built-in Styles on page 16-54 for style options.

The open tag syntax is as follows:

```
<OpenTag style = "style_value" width = "num_width">
```

The following table includes the built-in style options for the style attribute.

Built-in Styles

Style Value	Example
classic	<pre>/* single line comments */ /* * multiple line comments * second line */</pre>
classic_cpp	<pre>// single line comments // // multiple line comments // second line //</pre>
box	<pre>/* * * * */</pre>
box_cpp	<pre>/// /// /// /// ///</pre>
open_box	<pre>/* * * * */</pre>
open_box_cpp	<pre>/// /// /// /// /// // banner contents ///</pre>

File Banner. This section contains comments and tokens for use in generating a custom file banner. The file banner precedes any C or C++ code generated by the model. If you omit the file banner section from the CGT file, then no file banner emits to the generated code. The following section is the file banner section provided with the default CGT file, *matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt*.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom file banner section (optional)
%%
<FileBanner style="classic">
File: %<FileName>

Real-Time Workshop code generated for Simulink model %<ModelName>.

Model version                : %<ModelVersion>
Real-Time Workshop file version : %<RTWFileVersion>
Real-Time Workshop file generated on : %<RTWFileGeneratedOn>
TLC version                  : %<TLCVersion>
C/C++ source code generated on   : %<SourceGeneratedOn>
%<CodeGenSettings>
</FileBanner>
    
```

Summary of Tokens for File Banner Generation

FileName	Name of the generated file (for example, "rtwdemo_udt.c").
FileType	Either "source" or "header". Designates whether generated file is a .c or .cpp file or an .h file.
FileTag	Given file names file.c or .cpp and file.h; the file tags are "file_c" and "file_h", respectively.
ModelName	Name of generating model.
ModelVersion	Version number of model.
RTWFileVersion	Version number of <i>model.rtw</i> file.
RTWFileGeneratedOn	Timestamp of <i>model.rtw</i> file.

Summary of Tokens for File Banner Generation (Continued)

TLCVersion	Version of Target Language Compiler.
SourceGeneratedOn	Timestamp of generated file.
CodeGenSettings	Code generation settings for model: target language, target selection, embedded hardware selection, emulation hardware selection, code generation objectives (in priority order), and Code Generation Advisor validation result.

Function Banner. This section contains comments and tokens for use in generating a custom function banner. The function banner precedes any C or C++ function generated during the build process. If you omit the function banner section from the CGT file, the default function banner emits to the generated code. The following section is the default function banner section provided with the default CGT file, *matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt*.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom function banner section (optional)
%% Customize function banners by using the following predefined tokens:
%% %<ModelName>, %<FunctionName>, %<FunctionDescription>, %<Arguments>,
%% %<ReturnType>, %<GeneratedFor>, %<BlockDescription>.
%%
<FunctionBanner style="classic">
%<FunctionDescription>
%<BlockDescription>
</FunctionBanner>

```

Summary of Tokens for Function Banner Generation

FunctionName	Name of function
Arguments	List of function arguments
ReturnType	Return type of function
ModelName	Name of generating model
FunctionDescription	Short abstract about the function

Summary of Tokens for Function Banner Generation (Continued)

GeneratedFor	Full block path for the generated function
BlockDescription	<p>User input from the Block Description parameter of the block properties dialog box. BlockDescription contains an optional token attribute, style. The only valid value for style is content_only, which is case-sensitive and enclosed in double quotes. Use the content_only style when you want to include only the block description content that you entered in the block parameter dialog. The syntax for the token attribute style is:</p> <pre style="margin-left: 40px;">%<BlockDescription style = content_only ></pre>

Shared Utility Function Banner. The shared utility function banner section contains comments and tokens for use in generating a custom shared utility function banner. The shared utility function banner precedes any C or C++ shared utility function generated during the build process. If you omit the shared utility function banner section from the CGT file, the default shared utility function banner emits to the generated code. The following section is the default shared utility function banner section provided with the default CGT file, *matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt*.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom shared utility function banner section (optional)
%% Customize banners for functions generated in shared location by using the
%% following predefined tokens: %<FunctionName>, %<FunctionDescription>,
%% %<Arguments>, %<ReturnTypes>.
%%
<SharedUtilityBanner style="classic">
%<FunctionDescription>
</SharedUtilityBanner>

```

Summary of Tokens for Shared Utility Function Banner Generation

FunctionName	Name of function
Arguments	List of function arguments
ReturnType	Return type of function
FunctionDescription	Short abstract about function

File Trailer. The file trailer section contains comments for generating a custom file trailer. The file trailer follows any C or C++ code generated from the model. If you omit the file trailer section from the CGT file, no file trailer emits to the generated code. The following section is the default file trailer provided in the default CGT file.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom file trailer section (optional)
%%
<FileTrailer style="classic">
File trailer for Real-Time Workshop generated code.

[EOF]
</FileTrailer>

```

All of the tokens available for the file banner are available for the file trailer. See Summary of Tokens for File Banner Generation on page 16-55.

Template Symbols and Rules

Introduction

“Template Symbol Groups” on page 16-59 and “Template Symbols” on page 16-61 describe MPF template symbols and rules for using them. The location of a symbol in one of the supplied template files (code_c_template.cgt, code_h_template.cgt, data_c_template.cgt, or data_h_template.cgt) determines where the items associated with that symbol are located in the corresponding generated file. “Template Symbol Groups” on page 16-59 identifies the symbol groups, starting with the parent (“Base”) group, followed by the children of each parent. “Template Symbols” on page 16-61 lists the symbols alphabetically.

Template Symbol Groups

Symbol Group	Symbol Names in This Group
Base (Parents)	Declarations Defines Definitions Documentation Enums Functions Includes Types
Declarations	ExternalCalibrationLookup1D ExternalCalibrationLookup2D ExternalCalibrationScalar ExternalVariableScalar
Defines	LocalDefines LocalMacros
Definitions	FilescopeCalibrationLookup1D FilescopeCalibrationLookup2D FilescopeCalibrationScalar FilescopeVariableScalar GlobalCalibrationLookup1D GlobalCalibrationLookup2D GlobalCalibrationScalar GlobalVariableScalar

Symbol Group	Symbol Names in This Group
Documentation	Abstract Banner Created Creator Date Description FileName History LastModifiedDate LastModifiedBy ModelName ModelVersion ModifiedBy ModifiedComment ModifiedDate ModifiedHistory
	Notes ToolVersion
Functions	CFunctionCode
Types	This parent has no children.

Template Symbols

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
Abstract	Documentation	N/A	User-supplied description of the model or file. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
Banner	Documentation	N/A	Comments located near top of the file. Contains information that includes model and Real-Time Workshop versions, and date file was generated.
CFunctionCode	Functions	File	All of the C/C++ functions. Must be at the bottom of the template.
Created	Documentation	N/A	Date when model was created. From Created on field on Model Properties dialog box.
Creator	Documentation	N/A	User who created model. From Created by field on Model Properties dialog box.
Date	Documentation	N/A	Date file was generated. Taken from computer clock.
Declarations	Base		Data declaration of any signal or parameter. For example, <code>extern real_T globalvar;</code>
Defines	Base	File	Any necessary <code>#defines</code> of .h files.
Definitions	Base	File	Data definition of any signal or parameter.

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
Description	Documentation	N/A	Description of model. From Model description field on Model Properties dialog box.**
Documentation	Base	N/A	Comments about how to interpret the Real-Time Workshop generated files.
Enums	Base	File	Enumerated data type definitions.
ExternalCalibrationLookup1D	Declarations	External	***
ExternalCalibrationLookup2D	Declarations	External	***
ExternalCalibrationScalar	Declarations	External	***
ExternalVariableScalar	Declarations	External	***
FileName	Documentation	N/A	Name of the generated file.
FilescopeCalibrationLookup1D	Definitions	File	***
FilescopeCalibrationLookup2D	Definitions	File	***
FilescopeCalibrationScalar	Definitions	File	***
FilescopeVariableScalar	Definitions	File	***
Functions	Base	File	Generated function code.
GlobalCalibrationLookup1D	Definitions	Global	***
GlobalCalibrationLookup2D	Definitions	Global	***
GlobalCalibrationScalar	Definitions	Global	***
GlobalVariableScalar	Definitions	Global	***
History	Documentation	N/A	User-supplied revision history of the generated files. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
Includes	Base	File	#include preprocessor directives.
LastModifiedDate	Documentation	N/A	Date when model was last saved. From Last saved on field on Model Properties dialog box.
LastModifiedBy	Documentation	N/A	User who last saved model. From Last saved by field on Model Properties dialog box.
LocalDefines	Defines	File	#define preprocessor directives from code-generation data dictionary.
LocalMacros	Defines	File	C/C++ macros local to the file.
ModelName	Documentation	N/A	Name of the model.
ModelVersion	Documentation	N/A	Version number of the Simulink model.
ModifiedBy	Documentation	N/A	Name of user who last modified the model. From Model version field on Model Properties dialog box.
ModifiedComment	Documentation	N/A	Comment user enters in the Modified Comment field on the Log Change dialog box. See “Creating a Model Change History” in the Simulink documentation.
ModifiedDate	Documentation	N/A	Date model was last modified before code was generated.

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
ModifiedHistory	Documentation	N/A	Text from Modified history field on Model Properties dialog box.**
Notes	Documentation	N/A	User-supplied miscellaneous notes about the model or generated files. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
ToolVersion	Documentation	N/A	A list of the versions of the toolboxes used in generating the code.
Types	Base		Data types of generated code.

* All symbol names must be enclosed between %< >. For example, %<Functions>.

** This symbol can be used to add a comment to the generated files. See “Adding Global Comments” on page 16-5. The code generator places the comment in each generated file whose template has this symbol name. The code generator places the comment at the location that corresponds to where the symbol name is located in the template file.

*** The description can be deduced from the symbol name. For example, GlobalCalibrationScalar is a symbol that identifies a scalar. It contains data of global scope that you can calibrate .

Rules for Modifying or Creating a Template

The following are the rules for creating any MPF template. “Comparison of a Template and Its Generated File” on page 16-43 illustrates several of these rules.

- 1** Place a symbol on a template within the %< > delimiter. For example, the symbol named `Includes` should look like this on a template: %<Includes>. *Note that symbol names are case sensitive.*
- 2** Place a symbol on a template where desired. Its location on the template determines where the item associated with this symbol is located in the generated file. If no item is associated with it, the symbol is ignored.
- 3** Place a C/C++ statement outside of the %< > delimiter, and on a different line than a %< > delimiter, for that statement to appear in the generated file. For example, `#pragma message ("my text")` in the template results in `#pragma message ("my text")` at the corresponding location in the generated file. Note that the statement must be compatible with your C/C++ compiler.
- 4** Use the `.cgt` extension for every template filename. (“cgt” stands for code generation template.)
- 5** Note that %% \$Revision: 1.1.4.10.4.1 \$ appears at the top of the MathWorks supplied templates. This is for internal MathWorks use only. It does not need to be placed on a user-defined template and does not show in a generated file.
- 6** Place a comment on the template between `/* */` as in standard ANSI C⁴. This results in `/*comment*/` on the generated file.
- 7** Each MPF template must have all of the Base group symbols, in predefined order. They are listed in “Template Symbol Groups” on page 16-59. Each symbol in the Base group is a parent. For example, `Declarations` is a parent symbol.
- 8** Each symbol in a non-Base group is a child. For example, `LocalMacros` is a child.

4. ANSI[®] is a registered trademark of the American National Standards Institute, Inc.

- 9 Except for Documentation children, all children must be placed after their parent, before the next parent, and before the `FUNCTIONS` symbol.
- 10 Documentation children can be located before or after their parent in any order anywhere in the template.
- 11 If a non-Documentation child is missing from the template, the code generator places the information associated with this child at its parent location in the generated file.
- 12 If a Documentation child is missing from the template, the code generator omits the information associated with that child from the generated file.

Configuring the Placement of Data in Generated Code

To...	Select or Enter...
Specify whether data is to be defined in the generated source file or in a single separate header file	Select Auto , Data defined in source file , or Data defined in single separate source file for the Data definition parameter.
Specify whether data is to be declared in the generated source file or in a single separate header file	Select Auto , Data defined in source file , or Data defined in single separate source file for the Data declaration parameter.
Specify the <code>#include</code> file delimiter to be used in generated files that contain the <code>#include</code> preprocessor directive for mpt data objects	Select Auto , Data defined in source file , or Data defined in single separate source file for the #include file delimiter parameter.
Name the generated module using the same name as the model or a user-specified name	Select Not specified , Same as model , or User specified for the Module naming parameter.
Control whether signal data objects are to be declared as global data in the generated code	Enter an integer value for the Signal display level parameter.
Declare a parameter data object as tunable global data in the generated code	Enter an integer value for the Parameter tune level parameter.

For details about data placement, see Chapter 12, “Managing Placement of Data Definitions and Declarations”.

Ensuring Delimiter Is Specified for All #Includes

Understanding the purpose of this procedure requires understanding the `Header file` property of a data object, described in `Parameter and Signal Property Values` on page 6-2, and applied in “Creating mpt Data Objects with Data Object Wizard” on page 11-11. For a particular data object, you can specify as the `Header file` property value a `.h` filename where that data object will be declared. Then, in the `IncludeFile` section of the generated file, this `.h` file is indicated in a `#include` preprocessor directive.

Further, when specifying the filename as the `Header file` property value, you may or may not place it within the double-quote or angle-bracket delimiter. That is, you can specify it as `filename.h`, `"filename.h"`, or `<filename.h>`. The code generator finds every data object for which you specified a filename as its `Header file` property value *without* a delimiter. By default, it assigns to each of these the double-quote delimiter.

This procedure allows you to specify the angle-bracket delimiter for these instead of the default double-quote delimiter. See the figure below.

- 1** In the **#include file delimiter** field on the **Data Placement** pane of the Configuration Parameters dialog box, select `#include <header.h>` instead of the default `#include "header.h"`.
- 2** Click **Apply**.

Real-Time Workshop

Tools | Custom Code | Debug | Interface | Code Style | Templates | **Data Placement** |

Global data placement (custom storage classes only)

Data definition: Data defined in a single separate source file

Data definition filename: global.c

Data declaration: Data declared in a single separate header file

Data declaration filename: global.h

#include file delimiter: #include <header.h>

Global data placement (MPT data objects only)

Module naming: Not specified

Signal display level: 10 Parameter tune level: 10

Generate code only

Build

Revert Help Apply

Defining Model Configuration Variations

- “Introduction” on page 17-2
- “Viewing ERT Target Options in the Configuration Parameters Dialog Box or Model Explorer” on page 17-3

Introduction

This chapter explains how to use the Embedded Real-Time (ERT) target code generation options to configure models for production code generation. The discussion also includes other options that are not specific to the ERT target, but which affect ERT code generation.

Every model contains one or more named configuration sets that specify model parameters such as solver options, code generation options, and other choices. A model can contain multiple configuration sets, but only one configuration set is active at any time. A configuration set includes code generation options that affect Real-Time Workshop code generation in general, and options that are specific to a given target, such as the ERT target.

Configuration sets can be particularly useful in embedded systems development. By defining multiple configuration sets in a model, you can easily retarget code generation from that model. For example, one configuration set might specify the default ERT target with external mode support enabled for rapid prototyping, while another configuration set might specify the ERT-based target for Visual C++ to generate production code for deployment of the application. Activation of either configuration set fully reconfigures the model for the appropriate type of code generation.

Before you work with the ERT target options, you should become familiar with

- Configuration sets and how to view and edit them in the Configuration Parameters dialog box. The *Simulink User's Guide* document contains detailed information on these topics.
- Real-Time Workshop code generation options and the use of the System Target File Browser. The Real-Time Workshop documentation contains detailed information on these topics.

For descriptions of the Embedded Real-Time (ERT) target code generation options, see “Configuration Parameters” in the Real-Time Workshop Embedded Coder reference documentation.

Viewing ERT Target Options in the Configuration Parameters Dialog Box or Model Explorer

The Configuration Parameters dialog box and Model Explorer provide the quickest routes to a model's active configuration set. Illustrations throughout this chapter and "Configuration Parameters" in the Real-Time Workshop Embedded Coder reference documentation show the Configuration Parameters dialog box view of model parameters (unless otherwise noted).

Generating Code and Building Executables

- Chapter 18, “Generating Code Modules”
- Chapter 19, “Generating Reports for Code Reviews and Traceability Analysis”
- Chapter 20, “Optimizing Generated Code”
- Chapter 21, “Developing Models and Code That Comply with Industry Standards and Guidelines”

Generating Code Modules

Code Modules

In this section...

“Introduction” on page 18-2

“Generated Code Modules” on page 18-2

“User-Written Code Modules” on page 18-5

“Customizing Generated Code Modules ” on page 18-5

Introduction

This section summarizes the code modules and header files that make up a Real-Time Workshop Embedded Coder program and describes where to find the code modules and header files.

The easiest way to locate and examine the generated code files is to use the Real-Time Workshop Embedded Coder code generation report. The code generation report provides a table of hyperlinks that you click to view the generated code in the MATLAB Help browser. For more information, see “Generating an HTML Code Generation Report” on page 19-4.

Generated Code Modules

The Real-Time Workshop Embedded Coder software creates a build folder in your working folder to store generated source code. The build folder also contains object files, a makefile, and other files created during the code generation process. The default name of the build folder is *model_ert_rtw*.

The Real-Time Workshop® Embedded Coder™ File Packaging on page 18-3 section summarizes the structure of source code generated by the Real-Time Workshop Embedded Coder software.

Note The Real-Time Workshop Embedded Coder file packaging differs slightly (but significantly) from the file packaging employed by the GRT, GRT malloc, and other nonembedded targets. For more information, see the Real-Time Workshop documentation.

Real-Time Workshop Embedded Coder File Packaging

File	Description
<i>model.c</i> or <i>.cpp</i>	Contains entry points for code implementing the model algorithm (for example, <i>model_step</i> , <i>model_initialize</i> , and <i>model_terminate</i>).
<i>model_private.h</i>	Contains local macros and local data that are required by the model and subsystems. This file is included in the <i>model.c</i> file as a <code>#include</code> statement. You do not need to include <i>model_private.h</i> when interfacing handwritten code to the generated code of a model.
<i>model.h</i>	Declares model data structures and a public interface to the model entry points and data structures. Also provides an interface to the real-time model data structure (<i>model_M</i>) with accessor macros. <i>model.h</i> is included in the subsystem <i>.c</i> or <i>.cpp</i> files of the model. If you are interfacing your handwritten code to generated code for one or more models, include <i>model.h</i> for each of those models.
<i>model_data.c</i> or <i>.cpp</i> (conditional)	<i>model_data.c</i> or <i>.cpp</i> is conditionally generated. It contains the declarations for the parameters data structure, the constant block I/O data structure, and any zero representations for the model structure data types. If these data structures and zero representations are not used in the model, <i>model_data.c</i> or <i>.cpp</i> is not generated. These structures and zero representations are declared extern in <i>model.h</i> .
<i>model_types.h</i>	Provides forward declarations for the real-time model data structure and the parameters data structure. Function declarations of reusable functions might need these declarations. Also provides type definitions for user-defined types used by the model.
<i>rtwtypes.h</i>	Defines data types, structures, and macros required by Real-Time Workshop Embedded Coder generated code. Most other generated code modules also require these definitions.
<i>ert_main.c</i> or <i>.cpp</i> (optional)	If the Generate an example main program option is on, this file is generated. (This option is on by default.) See “Generate an example main program”.

Real-Time Workshop Embedded Coder File Packaging (Continued)

File	Description
autobuild.h (optional)	<p>If the Generate an example main program option is off, this file is generated. (See “Generate an example main program”.)</p> <p>autobuild.h contains <code>#include</code> directives required by the static version of the <code>ert_main.c</code> main program module. Because the static <code>ert_main.c</code> is not created at code generation time, <code>ert_main.c</code> includes <code>autobuild.h</code> to access model-specific data structures and entry points.</p> <p>For more information, see “Static Main Program Module” on page 31-16.</p>
<code>model_capi.c</code> or <code>.cpp</code> <code>model_capi.h</code> (optional)	<p>Provides data structures that enable a running program to access model signals, states, and parameters without external mode. To learn how to generate and use the <code>model_capi.c</code> or <code>.cpp</code> and <code>.h</code> files, see “Interacting with Target Application Data Using the C API” in the Real-Time Workshop documentation.</p>

You can customize the generated set of files in several ways:

- **File packaging formats:** Specify the number of source files generated for your model. In the Configuration Parameter dialog box, on the **Real-Time Workshop > Code Placement** pane, specify the **File packaging format** parameter. For more information, see “Customizing Generated Code Modules” on page 18-5.
- **Nonvirtual subsystem code generation:** Instruct the Real-Time Workshop software to generate separate functions, within separate code files, for any nonvirtual subsystems. You can control the names of the functions and of the code files. For further information, see “Creating Subsystems” in the Real-Time Workshop documentation.
- **Custom storage classes:** Use custom storage classes to partition generated data structures into different files based on file names that you specify. For further information, see Chapter 7, “Creating and Using Custom Storage Classes”.

- **Module Packaging Features (MPF):** Direct the generated code into a required set of `.c` or `.cpp` and `.h` files, and control the internal organization of the generated files. For details, see *Defining Data Representation and Storage for Code Generation* on page 1.

User-Written Code Modules

Code that you write to interface with generated model code usually includes a customized main module (based on a main program provided by the Real-Time Workshop software), and may also include interrupt handlers, device driver blocks and other S-functions, and other supervisory or supporting code.

Establish a working folder for your own code modules. Put your working folder on the MATLAB path. Minimally, you must also modify the ERT template makefile and system target file so that the build process can find your source and object files. If you want to generate code for a particular microprocessor or development board and deploy the code on target hardware with a cross-development system, make more extensive modifications to the ERT target files.

For information on how to customize the ERT target for your production requirements, see “Customizing Targets” in the Real-Time Workshop documentation.

Customizing Generated Code Modules

Real-Time Workshop Embedded Coder software provides a configuration parameter to specify how the generated source code is packaged into files. The configuration parameter “File packaging format” drop-down list options are located in the Configuration Parameter dialog box, on the **Real-Time Workshop > Code Placement** pane, in the Code Packaging section. The options are: **Modular**, **Compact (with separate data file)**, and **Compact. Generated Files According to File Packaging Format** on page 18-6 shows the files generated for each file packaging format and the files that have been removed.

Generated Files According to File Packaging Format

File Packaging Format	Generated Files	Removed Files
Modular (default)	<i>model.c</i> subsystem files (optional) <i>model.h</i> <i>model_types.h</i> <i>model_private.h</i> <i>model_data.c</i> (conditional)	None
Compact (with separate data file)	<i>model.c</i> <i>model.h</i> <i>model_data.c</i> (conditional)	<i>model_private.h</i> <i>model_types.h</i>
Compact	<i>model.c</i> <i>model.h</i>	<i>model_data.c</i> <i>model_private.h</i> <i>model_types.h</i>

The code generation process places the content of the removed files as follows:

Removed File	Generated Content In File
<i>model_private.h</i>	<i>model.c</i> and <i>model.h</i>
<i>model_types.h</i>	<i>model.h</i>
<i>model_data.c</i>	<i>model.c</i>

You can specify a different file packaging format for each referenced model.

If you specify **Utility function generation** as **Shared location** on the **Real-Time Workshop > Interface** pane of the Configuration Parameter dialog box, the code generation process generates separate files for utility

functions in a shared location, regardless of the file packaging format. If you specify the **Utility function generation** as Auto, the generated code for utility functions is dependent on the file packaging format as follows:

- Modular: Some shared utility files are in the build directory
- Compact (with separate data file): Utility functions are defined in *model.c*
- Compact: Utility functions are defined in *model.c*

File packaging formats, Compact or Compact (with separate data file) are not compatible with the following:

- A model with a subsystem, which is configured to generate separate source files
- A model with a noninlined S-function

Generating Reports for Code Reviews and Traceability Analysis

- “About HTML Code Generation Report Extensions” on page 19-2
- “Generating an HTML Code Generation Report” on page 19-4
- “Using the Code Interface Report to Analyze the Generated Code Interface” on page 19-6

About HTML Code Generation Report Extensions

The Real-Time Workshop Embedded Coder code generation report is an enhanced version of the HTML code generation report normally generated by the Real-Time Workshop build process. In the report:

- The **Summary** section lists version, date, and code generation objectives information. The **Configuration settings at the time of code generation** link opens a noneditable view of the Configuration Parameters dialog box that shows the Simulink model settings, including TLC options, at the time of code generation.
- The **Subsystem Report** section contains information on nonvirtual subsystems in the model.
- The **Code Interface Report** section provides information about the generated code interface, including model entry point functions and input/output data.
- The **Traceability Report** section allows you to account for **Eliminated / Virtual Blocks** that are untraceable, versus the listed **Traceable Simulink Blocks / Stateflow Objects / Embedded MATLAB Scripts**, providing a complete mapping between model elements and code.

In the **Generated Files** section of the **Contents** pane, you can click the names of source code files generated from your model to view their contents in a MATLAB Web browser window. In the displayed source code:

- The summary information is included as the code header.
- Global variable instances are hyperlinked to their definitions.
- If you selected the traceability option **Code-to-model**, hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click the hyperlinks to view the relevant blocks or subsystems in the Simulink model window. For more information about tracing code to blocks, see “Tracing Code to Model Objects Using Hyperlinks” on page 34-2.
- If you selected the traceability option **Model-to-code**, you can view the generated code for any block in the model. To highlight the generated code for a block in the HTML report, right-click the block and select **Real-Time Workshop > Navigate to Code**. For more information about tracking

blocks to generated code, see “Tracing Model Objects to Generated Code” on page 34-4.

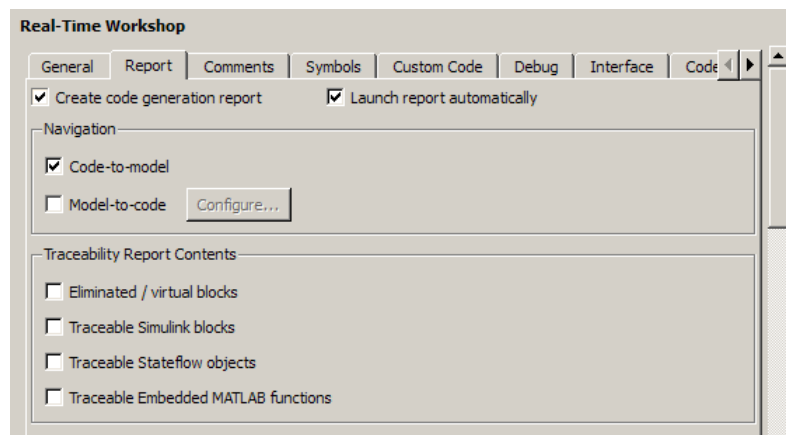
For a complete discussion on using traceability to verify generated code, see Chapter 34, “Verifying Generated Code”.

Generating an HTML Code Generation Report

To generate a Real-Time Workshop Embedded Coder code generation report,

- 1 With your ERT-based model open, open the Configuration Parameters dialog box or Model Explorer and navigate to the **Real-Time Workshop > Report** pane.
- 2 Select **Create code generation report** if it is not already selected. By default, **Launch report automatically** and **Code-to-model** also are selected, and **Model-to-code** is cleared, as shown in the figure below.

You can select or clear any of these options.



- 3 Generate code from your model or subsystem (for example, for a model, by clicking **Build** on the **Real-Time Workshop** pane of the Configuration Parameters dialog box).
- 4 The Real-Time Workshop build process writes the code generation report files in the `html` subdirectory of the build directory. The top-level HTML report file is named `model_codegen_rpt.html` or `subsystem_codegen_rpt.html`.
- 5 If you selected **Launch report automatically**, the Real-Time Workshop build process automatically opens a MATLAB Web browser window and displays the code generation report.

If you did not select **Launch report automatically**, you can open the code generation report (*model_codegen_rpt.html* or *subsystem_codegen_rpt.html*) manually into a MATLAB Web browser window, or into another Web browser.

- 6** If you selected **Code-to-model**, hyperlinks to blocks in the generating model are created in the report files. When you view the report files in a MATLAB Web browser, clicking on these hyperlinks displays and highlights the referenced blocks in the model. For more information, see “Tracing Code to Model Objects Using Hyperlinks” on page 34-2.
- 7** If you selected **Model-to-code**, model-to-code highlighting support is included in the generated HTML report. To highlight the generated code for a block in your Simulink model, right-click the block and select **Real-Time Workshop > Navigate to Code**. This selection highlights the generated code for the block in the HTML code generation report. For more information, see “Tracing Model Objects to Generated Code” on page 34-4 and “Customizing Traceability Reports” on page 34-7.

Notes

- For large models (containing over 1000 blocks), you may find that HTML report generation takes longer than you want. In this case, consider clearing the **Code-to-model** and **Model-to-code** check boxes. The report will be generated faster.
 - You can also view the HTML report files, as well as the generated code files, in the Simulink Model Explorer. See “Viewing Generated Code in the Model Explorer Code Viewer” in the Real-Time Workshop documentation for details.
-

Using the Code Interface Report to Analyze the Generated Code Interface

In this section...

“Code Interface Report Overview” on page 19-6

“Generating a Code Interface Report” on page 19-7

“Navigating Code Interface Report Subsections” on page 19-9

“Interpreting the Entry Point Functions Subsection” on page 19-10

“Interpreting the Inports and Outports Subsections” on page 19-13

“Interpreting the Interface Parameters Subsection” on page 19-14

“Interpreting the Data Stores Subsection” on page 19-16

“Code Interface Report Limitations” on page 19-17

Code Interface Report Overview

When you select the **Create code generation report** option for an ERT-based model, a **Code Interface Report** section is automatically included in the generated HTML report. The **Code Interface Report** section provides documentation of the generated code interface, including model entry point functions and interface data, for consumers of the generated code. The information in the report can help facilitate code review and code integration.

The code interface report includes the following subsections

- **Entry Point Functions** — interface information about each model entry point function, including `model_initialize`, `model_step`, and (if applicable) `model_terminate`
- **Inports and Outports** — interface information about each model inport and outport
- **Interface Parameters** — interface information about tunable parameters that are associated with the model
- **Data Stores** — interface information about global data stores and data stores with non-auto storage that are associated with the model

For limitations that apply to code interface reports, see “Code Interface Report Limitations” on page 19-17

Note This section uses the following demo models for illustration purposes:

- `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the demo model window) for examples of report subsections
 - `rtwdemo_mrmtbb` for examples of timing information
 - `rtwdemo_fcncnprotoctrl` for examples of function argument and return value information
-

Generating a Code Interface Report

To generate a code interface report for your model, perform the following steps.

- 1** Open your model, go to the **Real-Time Workshop** pane of the Configuration Parameters dialog box, and select `ert.tlc` or an ERT-based **System target file**, if one is not already selected.
- 2** Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the option **Create code generation report**, if it is not already selected. The `rtwdemo_basicsc`, `rtwdemo_mrmtbb`, and `rtwdemo_fcncnprotoctrl` demo models used in this section select every **Report** pane option by default, but selecting **Create code generation report** alone is sufficient to generate a **Code Interface Report** section in the HTML report.

Alternatively, you can programmatically select the option by issuing the following MATLAB command:

```
set_param(bdroot, 'GenerateReport', 'on')
```

If the **Report** pane option **Code-to-model** is selected, the generated report will contain hyperlinks to the model. You should leave this value selected unless you plan to use the report outside the MATLAB environment.

- 3 Build the model. If you selected the **Report** pane option **Launch report automatically**, the code generation report opens automatically after the build process completes. (Otherwise, you can launch it manually from within the model build directory.)
- 4 To display the code interface report for your model, go to the **Contents** pane of the HTML report and click the **Code Interface Report** link. For example, here is the generated code interface report for the demo model `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the demo model window).

Entry Point Functions

Function: [rtwdemo_basicsc_initialize](#)

Prototype	void rtwdemo_basicsc_initialize(void)
Description	Initialization entry point of generated code
Timing	Called once
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Function: [rtwdemo_basicsc_step](#)

Prototype	void rtwdemo_basicsc_step(void)
Description	Output entry point of generated code
Timing	Called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Inports

[-]

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1	input1	real32_T	1
<Root>/In2	input2	real32_T	1
<Root>/In3	input3	real32_T	1
<Root>/In4	input4	real32_T	1

Outputs

Block Name	Code Identifier	Data Type	Dimension
<Root>/Out1	output	real32_T	1

Interface Parameters

[+] ...

Data Stores

Data Store Source	Code Identifier	Data Type	Dimension
<Root>/Data_Store_Memory	mode	boolean_T	1

For help navigating the content of the code interface report subsections, see “Navigating Code Interface Report Subsections” on page 19-9. For help interpreting the content of the code interface report subsections, see the sections beginning with “Interpreting the Entry Point Functions Subsection” on page 19-10.

Navigating Code Interface Report Subsections

To help you navigate code interface descriptions, the code interface report provides collapse/expand tokens and hyperlinks, as follows:

- For any lengthy subsection, the report provides [-] and [+] symbols that allow you to collapse or expand that section. In the example in the previous section, the symbols are provided for the **Inports** and **Interface Parameters** sections.
- Several forms of hyperlink navigation are provided in the code interface report. For example,
 - The **Table of Contents** located at the top of the code interface report provides links to each subsection.
 - You can click on each function name to go to its declaration in *model.c*.
 - You can click on each function’s header file name to go to the header file source listing.
 - If you selected the **Report** pane option **Code-to-model** for your model, you can click hyperlinks for any of the following to go to the corresponding location in the model display:
 - Function argument
 - Function return value
 - Inport
 - Outport
 - Interface parameter (if the parameter source is a block)
 - Data store (if the data store source is a Data Store Memory block)

For general backward and forward navigation within the HTML code generation report, use the **Back** and **Forward** buttons above the **Contents** section in the upper left corner of the report.

Interpreting the Entry Point Functions Subsection

The **Entry Point Functions** subsection of the code interface report provides the following interface information about each model entry point function, including `model_initialize`, `model_step`, and (if applicable) `model_terminate`.

Field	Description
Function:	Lists the function name. You can click on the function name to go to its declaration in <code>model.c</code> .
Prototype	Displays the function prototype, including the function return value, name, and arguments.
Description	Provides a text description of the function's purpose in the application.
Timing	Describes the timing characteristics of the function, such as how many times the function is called, or if it is called periodically, at what time interval. For a multirate timing example, see the <code>rtwdemo_mrmtbb</code> report excerpt below.
Arguments	If the function has arguments, displays the number, name, data type, and Simulink description for each argument. If you selected the Report pane option Code-to-model for your model, you can click the hyperlink in the description to go to the block corresponding to the argument in the model display. For argument examples, see the <code>rtwdemo_fcnpctctrl</code> report excerpt below.

Field	Description
Return value	If the function has a return value, displays the return value data type and Simulink description. If you selected the Report pane option Code-to-model for your model, you can click the hyperlink in the description to go to the block corresponding to the return value in the model display. For a return value example, see the <code>rtwdemo_fcncnprotoctrl</code> report excerpt below.
Header file	Lists the name of the header file for the function. You can click on the header file name to go to the header file source listing.

For example, here is the **Entry Point Functions** subsection for the demo model `rtwdemo_basicsc`.

Entry Point Functions

Function: [rtwdemo_basicsc_initialize](#)

Prototype	void rtwdemo_basicsc_initialize(void)
Description	Initialization entry point of generated code
Timing	Called once
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Function: [rtwdemo_basicsc_step](#)

Prototype	void rtwdemo_basicsc_step(void)
Description	Output entry point of generated code
Timing	Called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

To illustrate how timing information might be listed for a multirate model, here are the **Entry Point Functions** and **Inports** subsections for the demo model `rtwdemo_mrmtbb`. This multirate, discrete-time, multitasking model contains Inport blocks 1 and 2, which specify 1-second and 2-second sample times, respectively. The sample times are constrained to the specified times

by the **Periodic sample time constraint** option on the **Solver** pane of the Configuration Parameters dialog box.

Entry Point Functions

Function: [rtwdemo_mrmtbb_initialize](#)

Prototype	void rtwdemo_mrmtbb_initialize(void)
Description	Initialization entry point of generated code
Timing	Called once
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: [rtwdemo_mrmtbb_step0](#)

Prototype	void rtwdemo_mrmtbb_step0(void)
Description	Output entry point of generated code
Timing	Called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: [rtwdemo_mrmtbb_step1](#)

Prototype	void rtwdemo_mrmtbb_step1(void)
Description	Output entry point of generated code
Timing	Called periodically, every 2 seconds
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

To illustrate how function arguments and return values are displayed in the report, here is the **Entry Point Functions** description of the model step function for the demo model `rtwdemo_fcncprotoctr1`.

Function: [rtwdemo_fcncnprotctrl_step_custom](#)

Prototype	boolean_T rtwdemo_fcncnprotctrl_step_custom (const real_T argIn1, const BusObject *const argIn2, BusObject *argOut2, const BusObject *const argIn3, uint8_T *argIn4)		
Description	Output entry point of generated code		
Timing	Called periodically, every 0.2 seconds		
Arguments	[-]		
	#	Name	Data Type
	1	argIn1	const real_T
	2	argIn2	const BusObject *const
	3	argOut2	BusObject *
	4	argIn3	const BusObject *const
	5	argIn4	uint8_T *
			Description
			<Root>/In1
			<Root>/In2
			<Root>/Out2
			<Root>/In3
			<Root>/In4
Return value	Data Type	Description	
	boolean_T	<Root>/Out1	
Header file	rtwdemo_fcncnprotctrl.h		

Interpreting the Inports and Outports Subsections

The **Inports** and **Outports** subsections of the code interface report provide the following interface information about each inport and output in the model.

Field	Description
Block Name	Displays the Simulink block name of the inport or output. If you selected the Report pane option Code-to-model for your model, you can click on each inport or output Block Name value to go to its location in the model display.
Code Identifier	Lists the identifier associated with the inport or output data in the generated code, as follows: <ul style="list-style-type: none"> • If the data is defined in the generated code, the field displays the identifier string. • If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier string prefixed with the label 'Imported data:'.

Field	Description
	<ul style="list-style-type: none"> If the data is neither defined nor declared in the generated code — for example, if the option Generate reusable code is selected for the model — the field displays the string 'Defined externally'.
Data Type	Lists the data type of the inport or outport.
Dimension	Lists the dimensions of the inport or outport (for example, 1 or [4, 5]).

For example, here are the **Inports** and **Outports** subsections for the demo model `rtwdemo_basicsc`.

Inports

[-]

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1	input1	real32_T	1
<Root>/In2	input2	real32_T	1
<Root>/In3	input3	real32_T	1
<Root>/In4	input4	real32_T	1

Outports

Block Name	Code Identifier	Data Type	Dimension
<Root>/Out1	output	real32_T	1

Interpreting the Interface Parameters Subsection

The **Interface Parameters** subsection of the code interface report provides the following interface information about tunable parameters that are associated with the model.

Field	Description
Parameter Source	<p>Lists the source of the parameter value, as follows:</p> <ul style="list-style-type: none"> • If the source of the parameter value is a block, the field displays the block name, such as <Root>/Gain2 or <S1>/Lookup1. If you selected the Report pane option Code-to-model for your model, you can click on the Parameter Source value to go to the parameter's location in the model display. • If the source of the parameter value is a workspace variable, the field displays the name of the workspace variable prefixed with the label 'Workspace variable:'. For example, <code>Workspace variable: K2</code>.
Code Identifier	<p>Lists the identifier associated with the tunable parameter data in the generated code, as follows:</p> <ul style="list-style-type: none"> • If the data is defined in the generated code, the field displays the identifier string. • If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier string prefixed with the label 'Imported data:'. • If the data is neither defined nor declared in the generated code — for example, if the option Generate reusable code is selected for the model — the field displays the string 'Defined externally'.
Data Type	Lists the data type of the tunable parameter.
Dimension	Lists the dimensions of the tunable parameter (for example, 1 or [4, 5, 6]).

For example, here is the **Interface Parameters** subsection for the demo model `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the demo model window).

Interface Parameters

[-]

Parameter Source	Code Identifier	Data Type	Dimension
Workspace variable: K2	K2	real_T	1
Workspace variable: LOWER	LOWER	real32_T	1
Workspace variable: T1Break	T1Break	real32_T	[1 11]
Workspace variable: T1Data	T1Data	real32_T	[1 11]
Workspace variable: T2Break	T2Break	real32_T	[1 3]
Workspace variable: T2Data	T2Data	real32_T	[3 3]
Workspace variable: UPPER	UPPER	real32_T	1
Workspace variable: K1	K1	int8_T	1

Interpreting the Data Stores Subsection

The **Data Stores** subsection of the code interface report provides the following interface information about global data stores and data stores with non-auto storage that are associated with the model.

Field	Description
Data Store Source	<p>Lists the source of the data store memory, as follows:</p> <ul style="list-style-type: none"> • If the data store is defined using a Data Store Memory block, the field displays the block name, such as <Root>/DS1. If you selected the Report pane option Code-to-model for your model, you can click on the Data Store Source value to go to the data store’s location in the model display. • If the data store is defined using a Simulink.Signal object, the field displays the name of the Simulink.Signal object prefixed with the label ‘Global:’.
Code Identifier	<p>Lists the identifier associated with the data store data in the generated code, as follows:</p> <ul style="list-style-type: none"> • If the data is defined in the generated code, the field displays the identifier string.

Field	Description
	<ul style="list-style-type: none"> • If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier string prefixed with the label 'Imported data:'. • If the data is neither defined nor declared in the generated code — for example, if the option Generate reusable code is selected for the model — the field displays the string 'Defined externally'.
Data Type	Lists the data type of the data store.
Dimension	Lists the dimensions of the data store (for example, 1 or [1, 2]).

For example, here is the **Data Stores** subsection for the demo model `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the demo model window).

Data Stores

Data Store Source	Code Identifier	Data Type	Dimension
<Root>/Data Store Memory	mode	boolean_T	1

Code Interface Report Limitations

The following limitations apply to code interface section of the HTML code generation reports.

- The code interface report does not support the GRT interface with an ERT target or the **C++ (Encapsulated)** language option. For these configurations, the code interface report will not be generated and will not appear in the HTML code generation report **Contents** pane.
- The code interface report supports data resolved with most custom storage classes (CSCs), except when the CSC properties are set in any of the following ways:

- The CSC property **Type** is set to `FlatStructure`. For example, the `BitField` and `Struct` CSCs in the Simulink package have **Type** set to `FlatStructure` and are not supported by the code interface report.
- The CSC property **Type** is set to `Other`. For example, the `GetSet` CSC in the Simulink package has **Type** set to `Other` and is not supported by the code interface report.
- The CSC property **Data access** is set to `Pointer`, indicating that imported symbols are declared as pointer variables rather than simple variables. This property is accessible only when the CSC property **Data scope** is set to `Imported` or `Instance-specific`.

In these cases, the report displays empty **Data Type** and **Dimension** fields.

- For outports, the code interface report cannot describe the associated memory (data type and dimensions) if the memory is optimized. In these cases, the report displays empty **Data Type** and **Dimension** fields.
- The code interface report does not support data type replacement using the **Real-Time Workshop > Data Type Replacement** pane of the Configuration Parameters dialog box. The data types listed in the report will link to built-in data types rather than their specified replacement data types.

Optimizing Generated Code

- “Configuring Production Code Optimizations” on page 20-2
- “Optimization Dependencies” on page 20-4
- “Optimizing Your Model with Configuration Wizard Blocks and Scripts” on page 20-6
- “Tips for Optimizing the Generated Code” on page 20-19

Configuring Production Code Optimizations

To...	Select or Specify...
Control whether parameter data for reusable subsystems is generated in a separate header file for each subsystem or in a single parameter data structure	Select Hierarchical or NonHierarchical for Parameter structure .
Replace multiply operations in array indices when accessing arrays in a loop	Select Simplify array indexing .
Store Boolean signals as one-bit bitfields instead of as a Boolean data type	Select Pack Boolean data into bitfields . Selecting this parameter enables the Bitfield declarator type specifier . To optimize your code further, select <code>uchar_T</code> , however this optimization benefit is dependent on your choice of target.
Pass each reusable subsystem output argument as an address of a local to reduce global memory usage and eliminate copying local variables back to global block I/O structures	Select Individual arguments for Pass reusable subsystem outputs as .
Generate initialization code for root-level inports and outports with a value of zero	Select Remove root level I/O zero initialization .
Generate additional code to set float and double storage explicitly to value 0.0	Select Use memset to initialize floats and doubles to 0.0 When you set this parameter, the <code>memset</code> function clears internal storage (regardless of type) to the integer bit pattern 0 (that is, all bits are off). The additional code generated when the option is off, is slightly less efficient. If the representation of floating-point zero used by your compiler and target CPU is identical to the integer bit pattern 0, you can gain efficiency by setting this parameter.
Suppress the generation of code that initializes internal work structures (for example, block states and block outputs) to zero	Select Remove internal state zero initialization .

To...	Select or Specify...
Generate run-time initialization code for a block that has states only if the block is in a system that can reset its states, such as an enabled subsystem	Select Optimize initialization code for model reference . This results in more efficient code, but requires that you not refer to the model from a Model block that resides in a system that resets its states. Such nesting results in an error. Turn this option off only if your application requires you refer to the model from Model blocks in systems that can reset their states.
Remove code that ensures that execution of the generated code produces the same results as simulation when out-of-range conversions occur	Select Remove code from floating-point to integer conversions that wraps out-of-range values . This reduces the size and increases the speed of the generated code at the cost of potentially producing results that do not match simulation in the case of out-of-range values.
Suppress generation of code that guards against fixed-point division by zero	Select Remove code that protects against division arithmetic exceptions . When you select this parameter, simulation results and results from generated code may no longer be in bit-for-bit agreement.
To minimize the amount of memory allocated for absolute and elapsed time counters	Specify an integer value for Application lifespan (days) . For more information on the allocation and operation of absolute and elapsed timers, see “Using Timers”, “Using Timers in Asynchronous Tasks”, and “Controlling Memory Allocation for Time Counters” in the Real-Time Workshop documentation.

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	

Optimizing Your Model with Configuration Wizard Blocks and Scripts

In this section...

“Overview” on page 20-6

“Adding a Configuration Wizard Block to Your Model” on page 20-8

“Using Configuration Wizard Blocks” on page 20-11

“Creating a Custom Configuration Wizard Block” on page 20-11

Overview

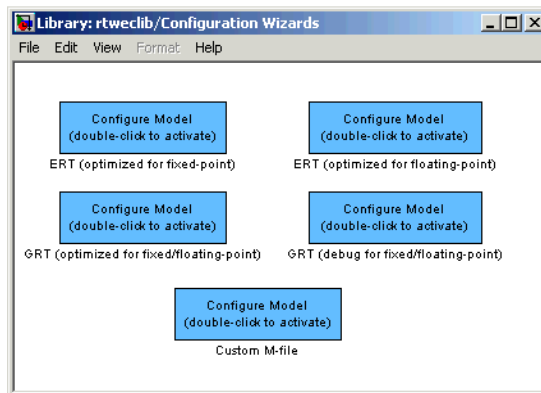
The Real-Time Workshop Embedded Coder software provides a library of *Configuration Wizard* blocks and scripts to help you configure and optimize code generation from your models quickly and easily.

The library provides a Configuration Wizard block you can customize, and four preset Configuration Wizard blocks.

Block	Description
Custom M-file	Automatically update active configuration parameters of parent model using a custom file
ERT (optimized for fixed-point)	Automatically update active configuration parameters of parent model for ERT fixed-point code generation
ERT (optimized for floating-point)	Automatically update active configuration parameters of parent model for ERT floating-point code generation

Block	Description
GRT (debug for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation with debugging enabled
GRT (optimized for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation

These are shown in the figure below.



When you add one of the preset Configuration Wizard blocks to your model and double-click it, a predefined MATLAB file script executes and configures all parameters of the model's active configuration set without manual intervention. The preset blocks configure the options optimally for one of the following cases:

- Fixed-point code generation with the ERT target
- Floating-point code generation with the ERT target
- Fixed/floating-point code generation with TLC debugging options enabled, with the GRT target.

- Fixed/floating-point code generation with the GRT target

The Custom block is associated with an example MATLAB file script that you can adapt to your requirements.

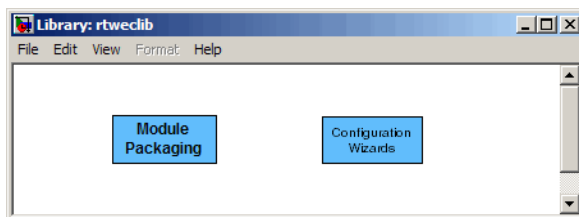
You can also set up the Configuration Wizard blocks to invoke the build process after configuring the model.

Adding a Configuration Wizard Block to Your Model

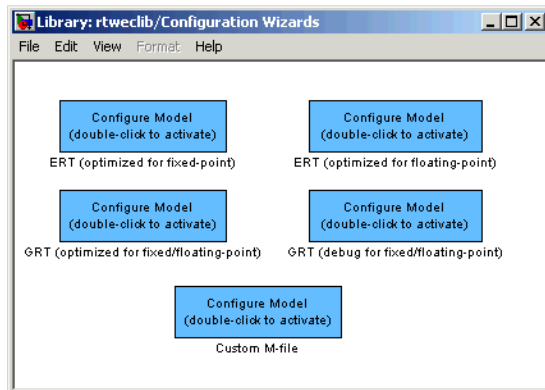
This section describes how to add one of the preset Configuration Wizard blocks to a model.

The Configuration Wizard blocks are available in the Real-Time Workshop Embedded Coder block library. To use a Configuration Wizard block:

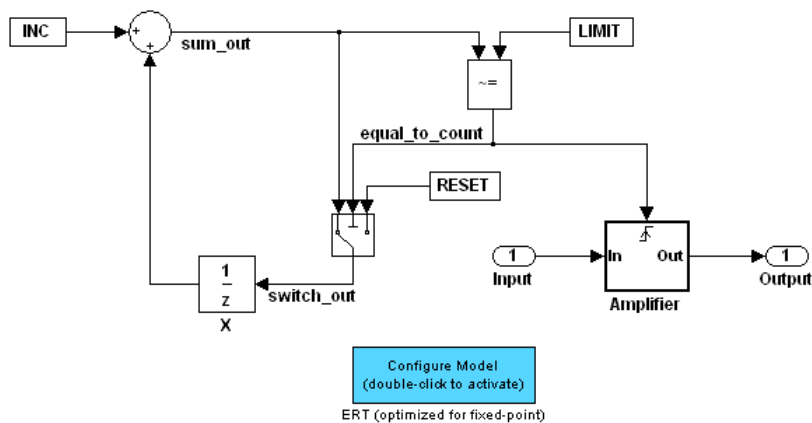
- 1** Open the model that you want to configure.
- 2** Open the Real-Time Workshop Embedded Coder library by typing the command `rtweclib`.
- 3** The top level of the library is shown below.



- 4** Double-click the Configuration Wizards icon. The Configuration Wizards sublibrary opens, as shown below.

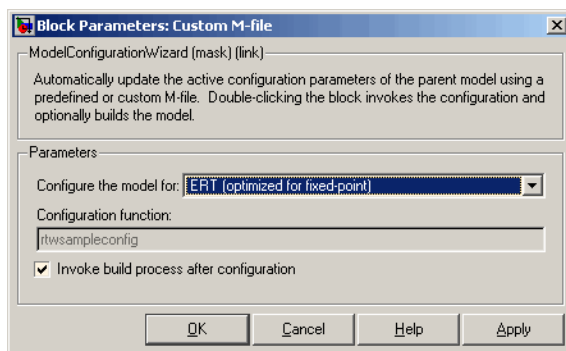


- 5 Select the Configuration Wizard block you want to use and drag and drop it into your model. In the figure below, the ERT (optimized for fixed-point) Configuration Wizard block has been added to the model.



- 6 You can set up the Configuration Wizard block to invoke the build process after executing its configuration script. If you do not want to use this feature, skip to the next step.

If you want the Configuration Wizard block to invoke the build process, right-click on the Configuration Wizard block in your model, and select **Mask Parameters...** from the context menu. Then, select the **Invoke build process after configuration** option, as shown below.



7 Click **Apply**, and close the Mask Parameters dialog box.

Note You should not change the **Configure the model for** option, unless you want to create a custom block and script. In that case, see “Creating a Custom Configuration Wizard Block” on page 20-11.

8 Save the model.

9 You can now use the Configuration Wizard block to configure the model, as described in the next section.

Using Configuration Wizard Blocks

Once you have added a Configuration Wizard block to your model, just double-click the block. The script associated with the block automatically sets all parameters of the active configuration set that are relevant to code generation (including selection of the appropriate target). You can verify that the options have changed by opening the Configuration Parameters dialog box and examining the settings.

If the **Invoke build process after configuration** option for the block was selected, the script also initiates the code generation and build process.

Note You can add more than one Configuration Wizard block to your model. This provides a quick way to switch between configurations.

Creating a Custom Configuration Wizard Block

The Custom Configuration Wizard block is shipped with an associated MATLAB file script, `rtwsampleconfig.m`. The script is located in the directory `matlabroot/toolbox/rtw/rtw`.

Both the block and the script are intended to provide a starting point for customization. This section describes:

- How to create a custom Configuration Wizard block linked to a custom script.

- Operation of the example script, and programming conventions and requirements for a customized script.
- How to run a configuration script from the MATLAB command line (without a block).

Setting Up a Configuration Wizard Block

This section describes how to set up a custom Configuration Wizard block and link it to a script. If you want to use the block in more than one mode, it is advisable to create a Simulink library to contain the block.

To begin, make a copy of the example script for later customization:

- 1** Create a directory to store your custom script. This directory should not be anywhere inside the MATLAB directory structure (that is, it should not be under *matlabroot*).

The discussion below refers to this directory as */my_wizards*.

- 2** Add the directory to the MATLAB path. Save the path for future sessions.

- 3** Copy the example script (*matlabroot/toolbox/rtw/rtw/rtwsampleconfig.m*) to the */my_wizards* directory you created in the previous steps. Then, rename the script as desired. The discussion below uses the name *my_configscript.m*.

- 4** Open the example script into the MATLAB editor. Scroll to the end of the file and enter the following line of code:

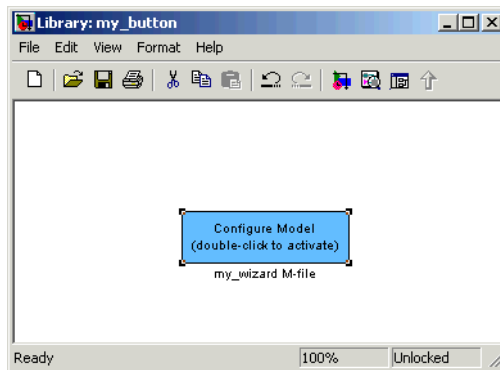
```
disp('Custom Configuration Wizard Script completed.');
```

This statement is used later as a test to verify that your custom block has executed the script.

- 5** Save your script and close the MATLAB editor.

The next step is to create a Simulink library and add a custom block to it. Do this as follows:

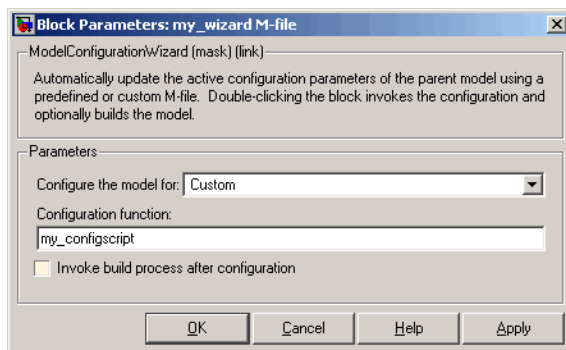
- 1 Open the Real-Time Workshop Embedded Coder library and the Configuration Wizards sublibrary, as described in “Adding a Configuration Wizard Block to Your Model” on page 20-8.
- 2 Select **New Library** from the **File** menu of the Configuration Wizards sublibrary window. An empty library window opens.
- 3 Select the Custom M-file block from the Configuration Wizards sublibrary and drag and drop it into the empty library window.
- 4 To distinguish your custom block from the original, edit the Custom M-file label under the block as desired.
- 5 Select **Save as** from the **File** menu of the new library window; save the library to the `/my_wizards` directory, under your library name of choice. In the figure below, the library has been saved as `my_button`, and the block has been labeled `my_wizard` M-file.



The next step is to link the custom block to the custom script:

- 1 Right-click on the block in your model, and select **Mask Parameters** from the context menu. Notice that the **Configure the model for** menu set to Custom. When Custom is selected, the **Configuration function** edit field is enabled, so you can enter the name of a custom script.
- 2 Enter the name of your custom script into the **Configuration function** field. (Do not enter the `.m` filename extension, which is implicit.) In the figure below, the script name `my_configscript` has been entered into the

Configuration function field. This establishes the linkage between the block and script.



- 3 Note that by default, the **Invoke build process after configuration** option is deselected. You can change the default for your custom block by selecting this option. For now, leave this option deselected.
- 4 Click **Apply** and close the Mask Parameters dialog box.
- 5 Save the library.
- 6 Close the Real-Time Workshop Embedded Coder library and the Configuration Wizards sublibrary. Leave your custom library open for use in the next step.

Now, test your block and script in a model. Do this as follows:

- 1 Open the vdp demo model by typing the command:

```
vdp
```

- 2 Open the Configuration Parameters dialog box and view the Real-Time Workshop options by clicking on the **Real-Time Workshop** entry in the list in the left pane of the dialog box.
- 3 Observe that the vdp demo is configured, by default, for the GRT target. Close the Configuration Parameters dialog box.
- 4 Select your custom block from your custom library. Drag and drop the block into the vdp model.
- 5 In the vdp model, double-click your custom block.
- 6 In the MATLAB window, you should see the test message you previously added to your script:

```
Custom Configuration Wizard Script completed.
```

This indicates that the custom block successfully executed the script.

- 7 Reopen the Configuration Parameters dialog box and view the **Real-Time Workshop** pane again. You should now see that the model is configured for the ERT target.

Before applying further edits to your custom script, proceed to the next section to learn about the operation and conventions of Configuration Wizard scripts.

Creating a Configuration Wizard Script

You should create your custom Configuration Wizard script by copying and modifying the example script, `rtwsampleconfig.m`. This section provides guidelines for modification.

The Configuration Function. The example script implements a single function without a return value. The function takes a single argument `cs`:

```
function rtwsampleconfig(cs)
```

The argument `cs` is a handle to a proprietary object that contains information about the model's active configuration set. The Simulink software obtains this handle and passes it in to the configuration function when the user double-clicks a Configuration Wizard block.

Your custom script should conform to this prototype. Your code should use `cs` as a “black box” object that transmits information to and from the active configuration set, using the accessor functions described below.

Accessing Configuration Set Options. To set options or obtain option values, use the Simulink `set_param` and `get_param` functions (if you are unfamiliar with these functions, see the Simulink Reference document).

Option names are passed in to `set_param` and `get_param` as strings specifying an *internal option name*. The internal option name is not always the same as the corresponding option label on the GUI (for example, the Configuration Parameters dialog box). The example configuration accompanies each `set_param` and `get_param` call with a comment that correlates internal option names to GUI option labels. For example:

```
set_param(cs, 'LifeSpan', '1'); % Application lifespan (days)
```

To obtain the current setting of an option in the active configuration set, call `get_param`. Pass in the `cs` object as the first argument, followed by the internal option name. For example, the following code excerpt tests the setting of the **Create code generation report** option:

```
if strcmp(get_param(cs, 'GenerateReport'), 'on')  
    ...
```

To set an option in the active configuration set, call `set_param`. Pass in the `cs` object as the first argument, followed by one or more parameter/value pairs that specify the internal option name and its value. For example, the following code excerpt turns off the **Support absolute time** option:

```
set_param(cs, 'SupportAbsoluteTime', 'off');
```

Selecting a Target. A Configuration Wizard script must select a target configuration. The example script uses the ERT target as a default. The script first stores string variables that correspond to the required **System target file**, **Template makefile**, and **Make command** settings:

```
stf = 'ert.tlc';
tmf = 'ert_default_tmf';
mc = 'make_rtw';
```

The system target file is selected by passing the `cs` object and the `stf` string to the `switchTarget` function:

```
switchTarget(cs, stf, []);
```

The template makefile and make command options are set by `set_param` calls:

```
set_param(cs, 'TemplateMakefile', tmf);
set_param(cs, 'MakeCommand', mc);
```

To select a target, your custom script needs only to set up the string variables `stf`, `tmf`, and `mc` and pass them to the appropriate calls, as above.

Obtaining Target and Configuration Set Information. The following utility functions and properties are provided so that your code can obtain information about the current target and configuration set, with the `cs` object:

- `isValidParam(cs, 'option')`: The `option` argument is an internal option name. `isValidParam` returns true if `option` is a valid option in the context of the active configuration set.
- `getPropEnabled(cs, 'option')`: The `option` argument is an internal option name. Returns true if this option is enabled (that is, writable).
- `IsERTTarget` property: Your code can detect whether or not the currently selected target is derived from the ERT target is selected by checking the `IsERTTarget` property, as follows:

```
isERT = strcmp(get_param(cs,'IsERTTarget'),'on');
```

This information can be used to determine whether or not the script should configure ERT-specific options, for example:

```
if isERT
    set_param(cs,'ZeroExternalMemoryAtStartup','off');
    set_param(cs,'ZeroInternalMemoryAtStartup','off');
    set_param(cs,'InitFltsAndDblsToZero','off');
    set_param(cs,'InlinedParameterPlacement',...
              'NonHierarchical');
    set_param(cs,'NoFixptDivByZeroProtection','on')
end
```

Invoking a Configuration Wizard Script from the MATLAB Command Prompt

Like any other MATLAB file, Configuration Wizard scripts can be run from the MATLAB command prompt. (The Configuration Wizard blocks are provided as a graphical convenience, but are not essential.)

Before invoking the script, you must open a model and instantiate a `cs` object to pass in as an argument to the script. After running the script, you can invoke the build process with the `rtwbuild` command. The following example opens, configures, and builds a model.

```
open my_model;
cs = getActiveConfigSet ('my_model');
rtwsampleconfig(cs);
rtwbuild('my_model');
```

Tips for Optimizing the Generated Code

In this section...

“Introduction” on page 20-19

“Using Configuration Wizard Blocks” on page 20-19

“Setting Hardware Implementation Parameters Correctly” on page 20-20

“Removing Unnecessary Initialization Code” on page 20-22

“Generating Pure Integer Code If Possible” on page 20-23

“Disabling MAT-File Logging” on page 20-23

“Using Virtualized Output Ports Optimization” on page 20-24

“Controlling Signal Storage” on page 20-25

“Using External Mode with the ERT Target” on page 20-26

Introduction

The Real-Time Workshop Embedded Coder software features a number of code generation options that can help you further optimize the generated code. This section highlights code generation options you can use to improve performance and reduce code size.

Most of the tips in this section apply specifically to the ERT target. See also the “Optimizing Generated Code” section of the Real-Time Workshop documentation for optimization techniques that are common to all target configurations.

Using Configuration Wizard Blocks

the Real-Time Workshop Embedded Coder software provides a library of *Configuration Wizard* blocks and scripts to help you configure and optimize code generation from your models quickly and easily.

When you add one of the preset Configuration Wizard blocks to your model and double-click it, an MATLAB file script executes and configures all parameters of the model’s active configuration set without user intervention.

The preset blocks configure the options optimally for common fixed- and floating-point code generation scenarios.

You can also create custom Configuration Wizard scripts and blocks.

See “Optimizing Your Model with Configuration Wizard Blocks and Scripts” on page 20-6 for detailed information.

Setting Hardware Implementation Parameters Correctly

Correct specification of target-specific characteristics of generated code (such as word sizes for char, short, int, and long data types, or desired rounding behaviors in integer operations) can be critical in embedded systems development. The **Hardware Implementation** category of options in a configuration set provides a simple and flexible way to control such characteristics in both simulation and code generation.

Before generating and deploying code, you should become familiar with the options on the **Hardware Implementation** pane of the Configuration Parameters dialog box. See “Hardware Implementation Pane” in the Simulink documentation and “Configuring the Hardware Implementation” in the Real-Time Workshop documentation for full details on the **Hardware Implementation** pane.

By configuring the **Hardware Implementation** properties of your model’s active configuration set to match the behaviors of your compiler and hardware, you can generate more efficient code. For example, if you specify the **Byte ordering** property, you can avoid generation of extra code that tests the byte ordering of the target CPU.

You can use the `rtwdemo_targetsettings` demo model to determine some implementation-dependent characteristics of your C or C++ compiler, as well as characteristics of your target hardware. By using this model in conjunction with your target development system and debugger, you can observe the behavior of the code as it executes on the target. You can then use this information to configure the **Hardware Implementation** parameters of your model.

To use this model, type the command

`rtwdemo_targetsettings`

Follow the instructions in the model window.

Removing Unnecessary Initialization Code

Consider selecting the **Remove internal state zero initialization** and **Remove root level I/O zero initialization** options on the **Optimization** pane.

These options (both off by default) control whether internal data (block states and block outputs) and external data (root inports and outputs whose value is zero) are initialized. Initializing the internal and external data whose value is zero is a precaution and may not be necessary for your application. Many embedded application environments initialize all RAM to zero at startup, making generation of initialization code redundant.

However, be aware that if you select **Remove internal state zero initialization**, it is not guaranteed that memory is in a known state each time the generated code begins execution. If you turn the option on, running a model (or a generated S-function) multiple times can result in different answers for each run.

This behavior is sometimes desirable. For example, you can turn on **Remove internal state zero initialization** if you want to test the behavior of your design during a warm boot (that is, a restart without full system reinitialization).

In cases where you have turned on **Remove internal state zero initialization** but still want to get the same answer on every run from a S-function generated by the Real-Time Workshop Embedded Coder software, you can use either of the following MATLAB commands before each run:

```
clear SFcnName
```

where *SFcnName* is the name of the S-function, or

```
clear mex
```

A related option, **Use memset to initialize floats and doubles**, lets you control the representation of zero used during initialization. See “Use memset to initialize floats and doubles to 0.0” in the Simulink reference documentation.

Note that the code still initializes data structures whose value is not zero when **Remove internal state zero initialization** and **Remove root level I/O zero initialization** are selected.

Note also that data of `ImportedExtern` or `ImportedExternPointer` storage classes is never initialized, regardless of the settings of these options.

Generating Pure Integer Code If Possible

If your application uses only integer arithmetic, deselect the **Support floating-point numbers** option in the **Software environment** section of the **Interface** pane to ensure that generated code contains no floating-point data or operations. When this option is deselected, an error is raised if any noninteger data or expressions are encountered during code generation. The error message reports the offending blocks and parameters.

Disabling MAT-File Logging

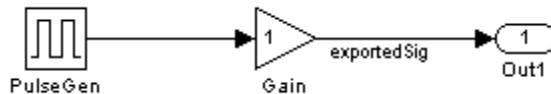
Clear the **MAT-file logging** option in the **Verification** section of the **Interface** pane. This setting is the default, and is recommended for embedded applications because it eliminates the extra code and memory usage required to initialize, update, and clean up logging variables. In addition to these efficiencies, clearing the **MAT-file logging** option lets you exploit further efficiencies under certain conditions. See “Using Virtualized Output Ports Optimization” on page 20-24 for information.

Note also that code generated to support MAT-file logging invokes `malloc`, which may be undesirable for your application.

Using Virtualized Output Ports Optimization

The *virtualized output ports* optimization lets you store the signal entering the root output port as a global variable. This eliminates code and data storage associated with root output ports when the **MAT-file logging** option is cleared and the TLC variable `FullRootOutputVector` equals 0, both of which are defaults for Real-Time Workshop Embedded Coder targets.

To illustrate this feature, consider the model shown in the following block diagram. Assume that the signal `exportedSig` has `exportedGlobal` storage class.



In the default case, the output of the Gain block is written to the signal storage location, `exportedSig`. No code or data is generated for the `Out1` block, which has become, in effect, a virtual block. This is shown in the following code fragment.

```

/* Gain Block: <Root>/Gain */
exportedSig = rtb_PulseGen * VirtOutPortLogOFF_P.Gain_Gain;
  
```

In cases where either the **MAT-file logging** option is enabled, or `FullRootOutputVector = 1`, the generated code represents root output ports as members of an external outputs vector.

The following code fragment was generated from the same model shown in the previous example, but with **MAT-file logging** enabled. The output port is represented as a member of the external outputs vector `VirtOutPortLogON_Y`. The Gain block output value is copied to both `exportedSig` and to the external outputs vector.

```

/* Gain Block: <Root>/Gain */
exportedSig = rtb_PulseGen * VirtOutPortLogON_P.Gain_Gain;

/* Outputport Block: <Root>/Out1 */
VirtOutPortLogON_Y.Out1 = exportedSig;
  
```

The overhead incurred by maintenance of data in the external outputs vector can be significant for smaller models being used to perform benchmarks.

Note that you can force root output ports to be stored in the external outputs vector (regardless of the setting of **MAT-file logging**) by setting the TLC variable `FullRootOutputVector` to 1. You can do this by adding the statement

```
%assign FullRootOutputVector = 1
```

to the Real-Time Workshop Embedded Coder system target file. Alternatively, you can enter the assignment with **TLC options** on the **Real-Time Workshop** pane of the Configuration Parameters dialog box.

For more information on how to control signal storage in generated code, see the “Defining Data Representation and Storage for Code Generation” section of the Real-Time Workshop documentation.

Controlling Signal Storage

the Real-Time Workshop software offers a number of options that let you control how signals in your model are stored and represented in the generated code. You can control whether signal storage is declared in global memory space, or locally in functions (that is, in stack variables).

For a complete discussion of signal storage options, see the “Defining Data Representation and Storage for Code Generation” section of the Real-Time Workshop documentation.

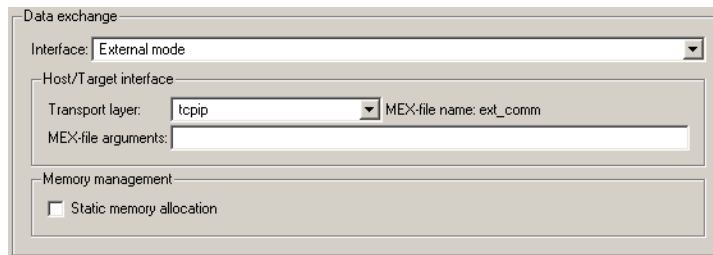
If you want to store signals in stack space, you must turn the **Enable local block outputs** option on. To do this

- 1 Select the **Optimization** pane of the Configuration Parameters dialog box. Make sure that **Signal storage reuse** is selected. If **Signal storage reuse** is cleared, **Enable local block outputs** is not available.
- 2 Select the **Enable local block outputs** option. Click **Apply** if necessary.

Using External Mode with the ERT Target

Selecting the **External mode** option turns on generation of code to support external mode communication between host (Simulink) and target systems. The Real-Time Workshop Embedded Coder software supports all features of Simulink external mode, as described in the “Communicating With Code Executing on a Target System Using Simulink External Mode” section of the Real-Time Workshop documentation.

This section discusses external mode options that may be of special interest to embedded systems designers. The next figure shows the **Data Exchange** subpane of the Configuration Parameters dialog box, **Interface** pane, with **External mode** selected.



Memory Management

Consider the **Memory management** option **Static memory allocation** before generating external mode code for an embedded target. Static memory allocation is generally desirable, as it reduces overhead and promotes deterministic performance.

When you select the **Static memory allocation** option, static external mode communication buffers are allocated in the target application. When **Static memory allocation** is deselected, communication buffers are allocated dynamically (with `malloc`) at run time.

Generation of Pure Integer Code with External Mode

The Real-Time Workshop Embedded Coder software supports generation of pure integer code when external mode code is generated. To do this, select the **External mode** option, and deselect the **Support floating-point numbers** option in the **Software environment** section of the **Interface** pane.

This enhancement lets you generate external mode code that is free of any storage definitions of double or float data type, and allows your code to run on integer-only processors

If you intend to generate pure integer code with **External mode** on, note the following requirements:

- All trigger signals must be of data type `int32`. Use a Data Type Conversion block if needed.
- When pure integer code is generated, the simulation stop time specified in the **Solver** options is ignored. To specify a stop time, run your target application from the MATLAB command line and use the `-tf` option. (See “Running the External Program” in the “External Mode” section of the Real-Time Workshop documentation.) If you do not specify this option, the application executes indefinitely (as if the stop time were `inf`).

When executing pure integer target applications, the stop time specified by the `-tf` command line option is interpreted as the number of base rate ticks to execute, rather than as an elapsed time in seconds. The number of ticks is computed as

$$\text{stop time in seconds} / \text{base rate step size in seconds}$$

Developing Models and Code That Comply with Industry Standards and Guidelines

- “What Are the Standards and Guidelines?” on page 21-2
- “Developing Models and Code That Comply with MAAB Guidelines” on page 21-4
- “Developing Models and Code That Comply with MISRA C Guidelines” on page 21-5
- “Developing Models and Code That Comply with the IEC 61508 Standard” on page 21-6
- “Developing Models and Code That Comply with the DO-178B Standard” on page 21-8

What Are the Standards and Guidelines?

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

Standard or Guidelines	Organization	For More Information, See...
Guidelines: Use of MATLAB, Simulink, and Stateflow software for control algorithm modeling – MathWorks Automotive Advisory Board (MAAB) Guidelines	MAAB	<ul style="list-style-type: none"> • Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow Software: PDF, Word • “Developing Models and Code That Comply with MAAB Guidelines” on page 21-4
Guidelines: Use of the C Language in Critical Systems (MISRA C ⁵)	Motor Industry Software Reliability Association (MISRA)	<ul style="list-style-type: none"> • MISRA C Web site • Technical Solution 1-1IFP0W on the MathWorks Web site • “Developing Models and Code That Comply with MISRA C Guidelines” on page 21-5
Standard: AUTomotive Open System ARchitecture (AUTOSAR)	AUTOSAR Development Partnership	<ul style="list-style-type: none"> • Publications and specifications available from the AUTOSAR Web site • Technical Solution 1-2WFS27 on the MathWorks Web site • Chapter 22, “Generating Code for AUTOSAR Software Components”

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

Standard or Guidelines	Organization	For More Information, See...
Standard: IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems	International Electrotechnical Commission	<ul style="list-style-type: none"> • IEC functional safety zone Web site • Model-Based Design for IEC 61508 (Excerpts) — For the complete document, see Technical Solution 1-32COJP on the MathWorks Web site. • “Developing Models and Code That Comply with the IEC 61508 Standard” on page 21-6
Standard: DO-178B, Software Considerations in Airborne Systems and Equipment Certification	Radio Technical Commission for Aeronautics (RTCA)	<ul style="list-style-type: none"> • Model-Based Design for DO-178B (Excerpts) — For the complete document, see Technical Solution 1-1ZLDDE on the MathWorks Web site. • “Developing Models and Code That Comply with the DO-178B Standard” on page 21-8

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

Developing Models and Code That Comply with MAAB Guidelines

The MathWorks Automotive Advisory Board (MAAB) involves major automotive OEMs and suppliers in the process of evolving MathWorks controls, simulation, and code generation products, including Simulink, Stateflow, and Real-Time Workshop. An important result of the MAAB has been the MAAB Guidelines.

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

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

Developing Models and Code That Comply with MISRA C Guidelines

The Motor Industry Software Reliability Association (MISRA⁶) has established “Guidelines for the Use of the C Language in Critical Systems” (MISRA C). For general information about MISRA C, see www.misra-c.com.

You can check that your Simulink model or subsystem is developed to increase the likelihood of generating MISRA-C:2004 compliant code for embedded applications. To check your model or subsystem, open the Simulink Model Advisor. Navigate to **By Product > Real-Time Workshop Embedded Coder** and run the following checks:

- “Check for blocks not recommended for MISRA-C:2004 compliance”
- “Check configuration parameters for MISRA-C:2004 compliance”

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

For information about using Real-Time Workshop Embedded Coder software within MISRA C guidelines, see Technical Solution 1-1IFP0W on the MathWorks Web site.

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

Developing Models and Code That Comply with the IEC 61508 Standard

In this section...

“Applying Simulink and Real-Time Workshop Technology to the IEC 61508 Standard” on page 21-6

“Checking for IEC 61508 Standard Compliance Using the Model Advisor” on page 21-6

“Validating Traceability” on page 21-9

Applying Simulink and Real-Time Workshop Technology to the IEC 61508 Standard

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

Checking for IEC 61508 Standard Compliance Using the Model Advisor

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

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

Validating Traceability

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

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

Developing Models and Code That Comply with the DO-178B Standard

In this section...

“Applying Simulink and Real-Time Workshop Technology to the DO-178B Standard” on page 21-8

“Checking for Standard Compliance Using the Model Advisor” on page 21-8

“Validating Traceability” on page 21-9

Applying Simulink and Real-Time Workshop Technology to the DO-178B Standard

Applying Model-Based Design successfully to a safety-critical system, requires extra consideration and rigor to ensure the system adheres to defined safety standards. DO-178B, Software Considerations in Airborne Systems and Equipment Certification, is such a standard. Because the standard was published when most software was coded by hand, the standard needs to be mapped to Model-Based Design technologies. Model-Based Design for DO-178B (Excerpts) provides a sampling of information available from a document that offers recommendations on how to apply Simulink, Real-Time Workshop, and third-party products for Model-Based Design to DO-178B measures and techniques. For the complete version of Model-Based Design for DO-178B, see Technical Solution 1-1ZLDDE on the MathWorks Web site.

Checking for Standard Compliance Using the Model Advisor

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

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

Validating Traceability

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

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

Generating Code for AUTOSAR Software Components

- “Overview of AUTOSAR Support” on page 22-2
- “Simulink Modeling Patterns for AUTOSAR” on page 22-3
- “Workflow for AUTOSAR ” on page 22-28
- “Importing an AUTOSAR Software Component” on page 22-30
- “Using the Configure AUTOSAR Interface Dialog Box” on page 22-32
- “Configuring Ports for Basic Software and Error Status Receivers” on page 22-37
- “Configuring Client-Server Communication” on page 22-39
- “Configuring Calibration Parameters” on page 22-48
- “Modifying and Validating an Existing AUTOSAR Interface” on page 22-49
- “Exporting a Single Runnable AUTOSAR Software Component” on page 22-50
- “Configuring Multiple Runnables” on page 22-55
- “Configuring AUTOSAR Options Programmatically” on page 22-58
- “Verifying the AUTOSAR Code with SIL and PIL Simulations” on page 22-59
- “Limitations and Tips” on page 22-62
- “Demos and Further Reading” on page 22-66

Overview of AUTOSAR Support

Real-Time Workshop Embedded Coder software supports AUTOSAR (*AUTomotive Open System ARchitecture*), an open and standardized automotive software architecture. AUTOSAR is developed jointly by automobile manufacturers, suppliers, and tool developers.

The AUTOSAR standard addresses:

- Architecture – Three layers, *Application*, *Runtime Environment (RTE)*, and *Basic Software*, enable decoupling of AUTOSAR Software Components from the execution platform. Standard interfaces between AUTOSAR Software Components and the Runtime Environment allow reuse or relocation of components within the Electronic Control Unit (ECU) topology of a vehicle.
- Methodology – Specification of code formats and description file templates, for example.
- Application Interfaces – Specification of interfaces for typical automotive applications.

For details on the AUTOSAR standard, go to www.autosar.org.

In Simulink, you can model AUTOSAR Software Components and related concepts. See “Simulink Modeling Patterns for AUTOSAR” on page 22-3.

Using Real-Time Workshop Embedded Coder software, you can generate AUTOSAR-compliant code and description files. See “Workflow for AUTOSAR” on page 22-28.

Simulink Modeling Patterns for AUTOSAR

In this section...

“AUTOSAR Software Components” on page 22-3

“AUTOSAR Communication” on page 22-11

“Calibration Parameters” on page 22-19

“Inter-Runnable Variables” on page 22-19

“Data Types” on page 22-20

“AUTOSAR Terminology” on page 22-25

This section describes how you model AUTOSAR Software Components and related concepts in Simulink.

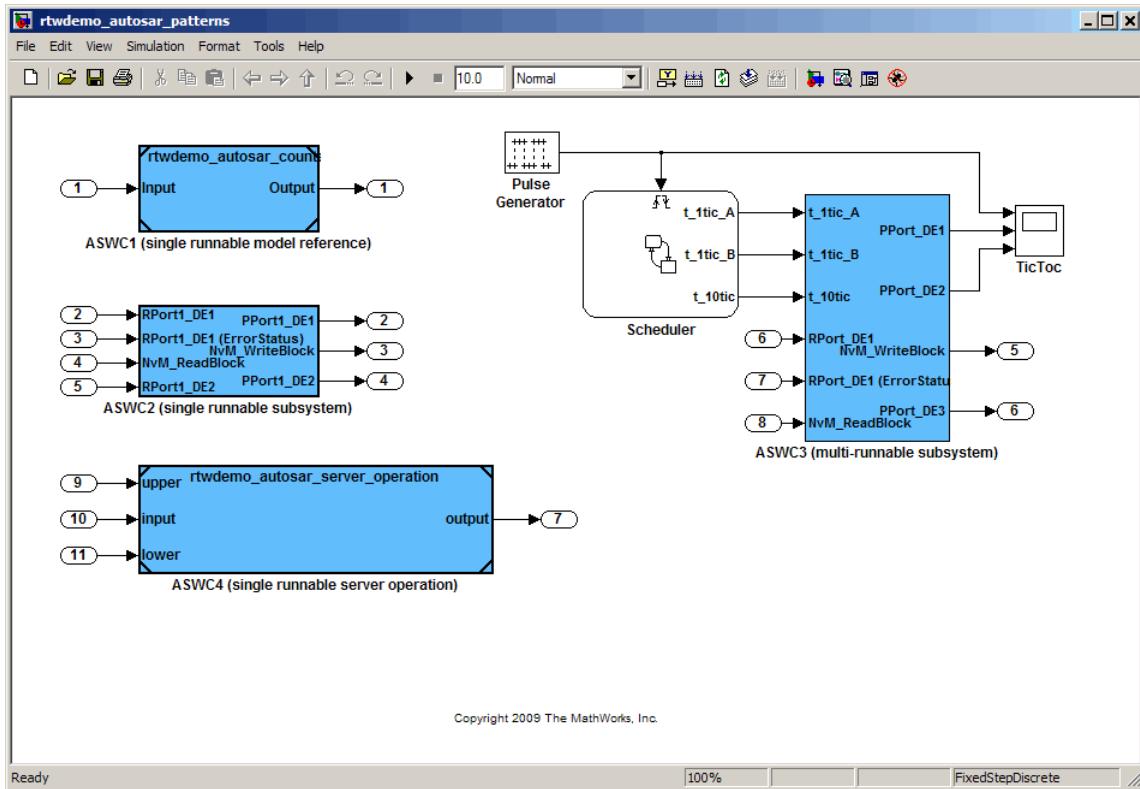
AUTOSAR Software Components

In AUTOSAR, application software consists of separate units, *AUTOSAR Software Components*.

Note An AUTOSAR Software Component is sometimes referred to as *atomic* because it is never split across more than one Electronic Control Unit (ECU). Do not confuse *atomic* in this context with the concept of Simulink atomic subsystems.

The behavior of an AUTOSAR Software Component is implemented by a single or multiple *runnable entities* (runnables), which expose well-defined connection points, *ports*.

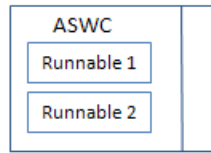
In Simulink, you can represent an AUTOSAR Software Component using a model or a subsystem. For example, the following figure shows modeling patterns for AUTOSAR Software Components (ASWC) labeled ASWC1, ASWC2, ASWC3, and ASWC4.



Runnables

AUTOSAR Software Components contain runnables that are directly or indirectly scheduled by the underlying AUTOSAR operating system.

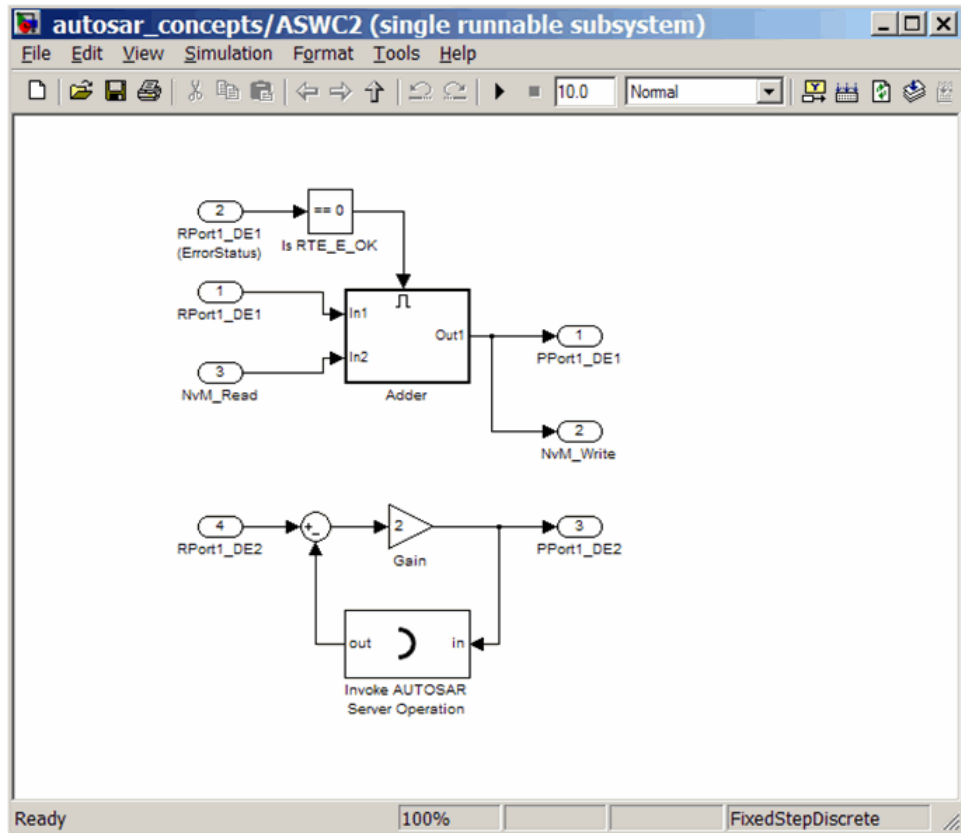
The following figure shows an AUTOSAR Software Component with two runnables, Runnable 1 and Runnable 2. Each runnable is triggered by RTEEvents, events generated by the AUTOSAR Runtime Environment (RTE). For example, TimingEvent is an RTEEvent that is generated periodically.



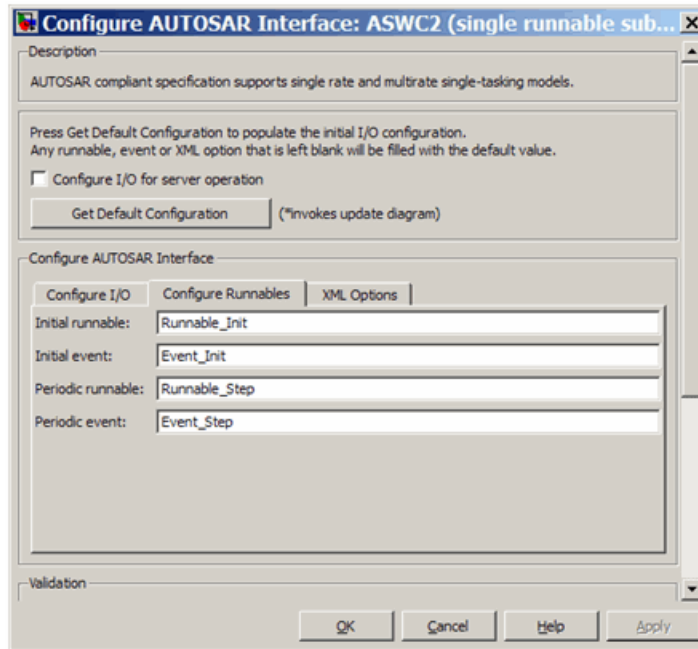
The components ASWC1, ASWC2 and ASWC4 contain single runnables. These components are represented by a subsystem or a model, and can be single- or multirate. However, the software implements each component as a single-tasking operation.

Note The software generates an additional runnable for the initialization function regardless of the modeling pattern.

ASWC2 is modeled as a single-rate, single-tasking atomic subsystem.

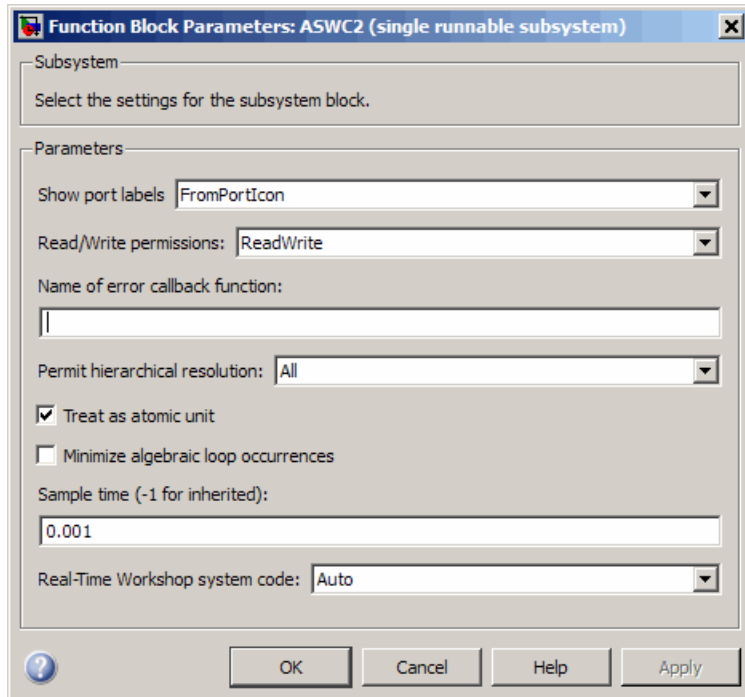


You can generate the ASWC2 runnable, which corresponds to the step function of the subsystem. Use the Configure AUTOSAR Interface dialog box to specify the names of the initial and periodic runnables, as shown by the following figure.

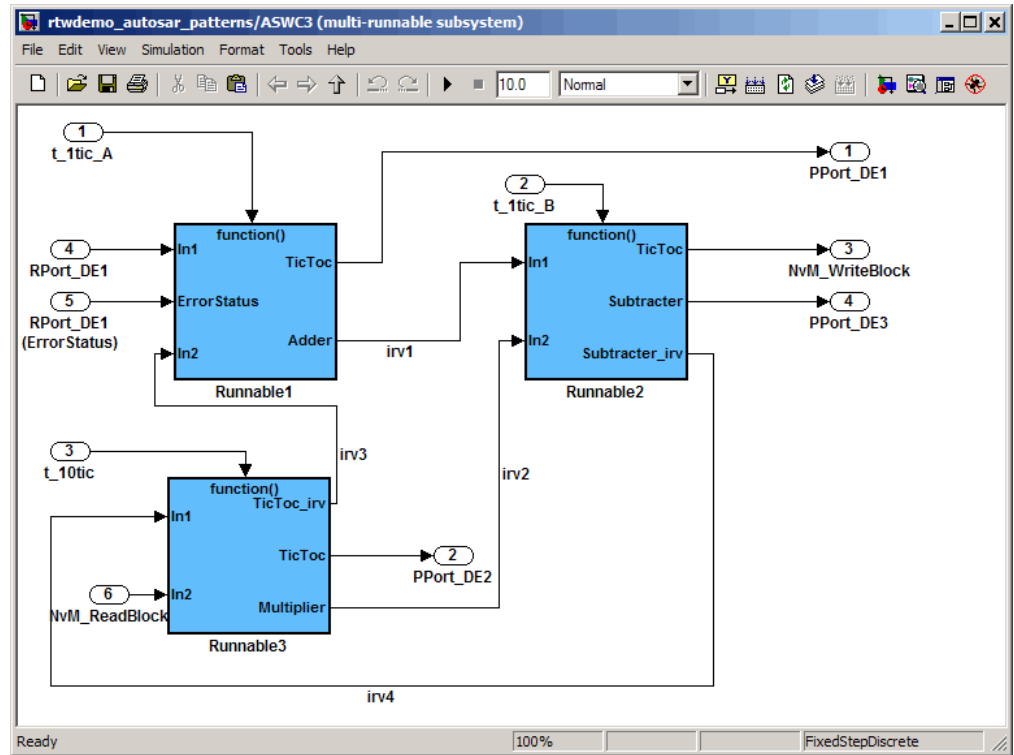


The software generates TimingEvents for the runnables, with the following periods:

- 0 for initial runnable.
- Fundamental sample time of model or atomic subsystem for periodic runnable. You specify this sample time in the **Sample time (-1 for inherited)** field in the Subsystem Parameters dialog box, as shown by the following figure.

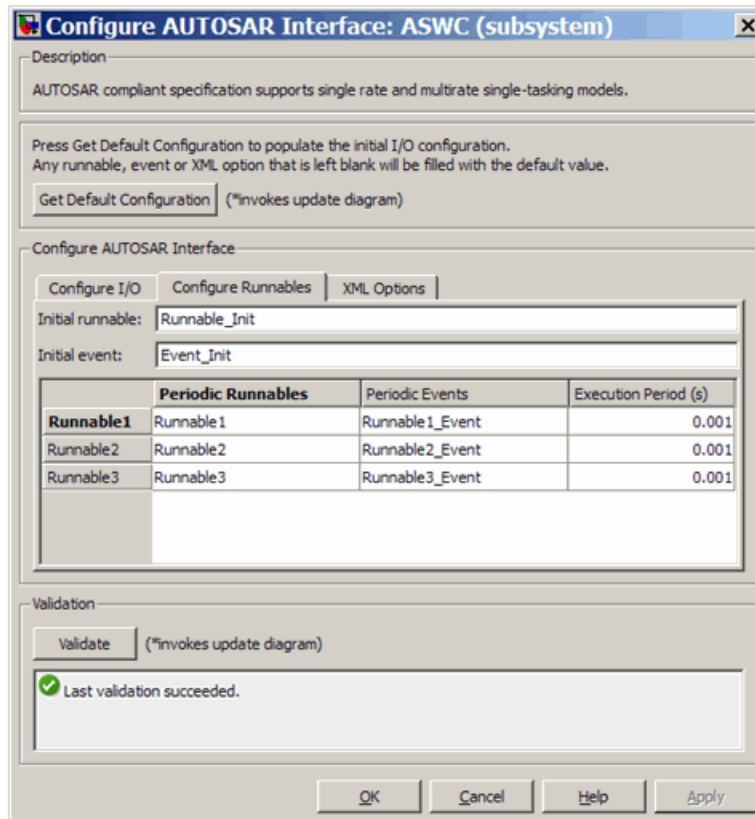


The component ASWC3 contains multiple runnables.



You can use the Export Functions feature to map the runnables to Simulink function-call subsystems. See “Configuring Multiple Runnables” on page 22-55. The software also generates an initialization runnable for the initialization function.

Use the Configure AUTOSAR Interface dialog box to specify the names of the multiple runnables and the periods of Timing Events.



Multiple Instantiation

AUTOSAR supports multiple instantiations of software components. However, Simulink supports multiple instantiations (reentrant code) only if a model is configured as a server operation. See “Configuring a Server Operation” on page 22-40.

To generate reentrant code for a model configured as a server operation, on the **Real-Time Workshop Interface** pane, select the **Generate reusable code** check box.

AUTOSAR Communication

AUTOSAR Software Components provide well-defined connection points, ports. There are two types of AUTOSAR ports:

- Require
- Provide

In addition, these AUTOSAR ports can reference two kinds of interfaces:

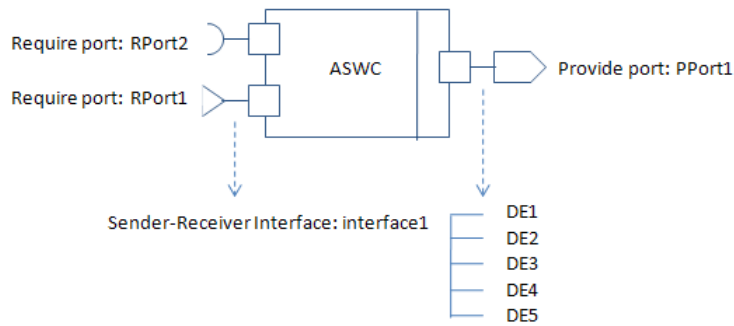
- Sender-Receiver
- Client-Server

The following figure shows an AUTOSAR Software Component with four ports representing all port and interface combinations.



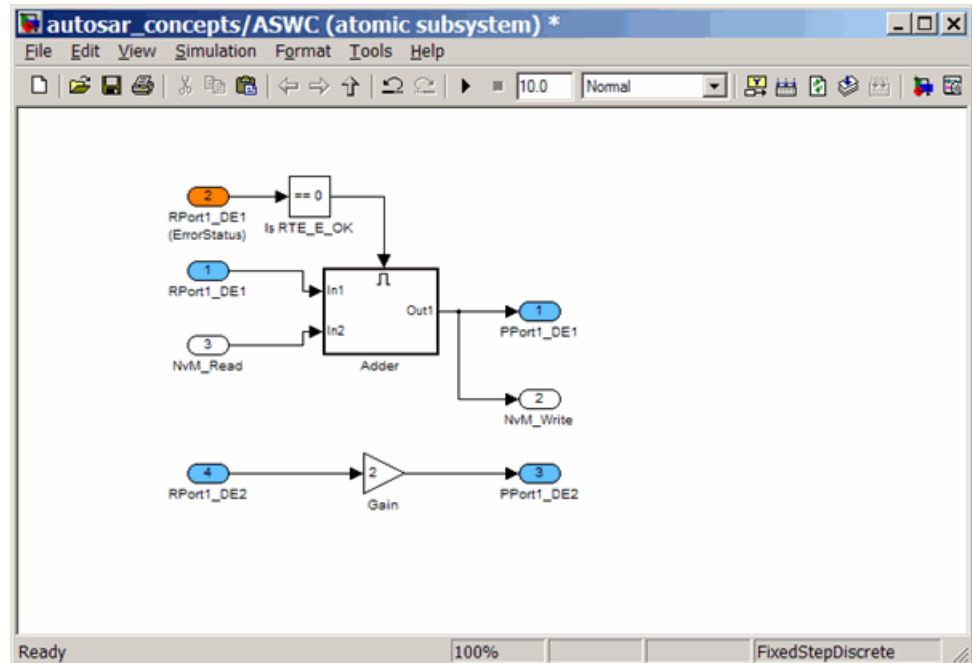
Sender-Receiver Interface

A Sender-Receiver Interface consists of one or more data elements. Although a Require or Provide port may reference a Sender-Receiver Interface, the AUTOSAR Software Component does not necessarily access all the data elements. For example, consider the following figure.



The AUTOSAR Software Component has a Require and Provide port that references the same Sender-Receiver Interface, `interface1`. Although this interface contains data elements `DE1`, `DE2`, `DE3`, `DE4`, and `DE5`, the component does not utilize all the data elements.

The following figure is an example of how you model, in Simulink, an AUTOSAR Software Component that accesses data elements.

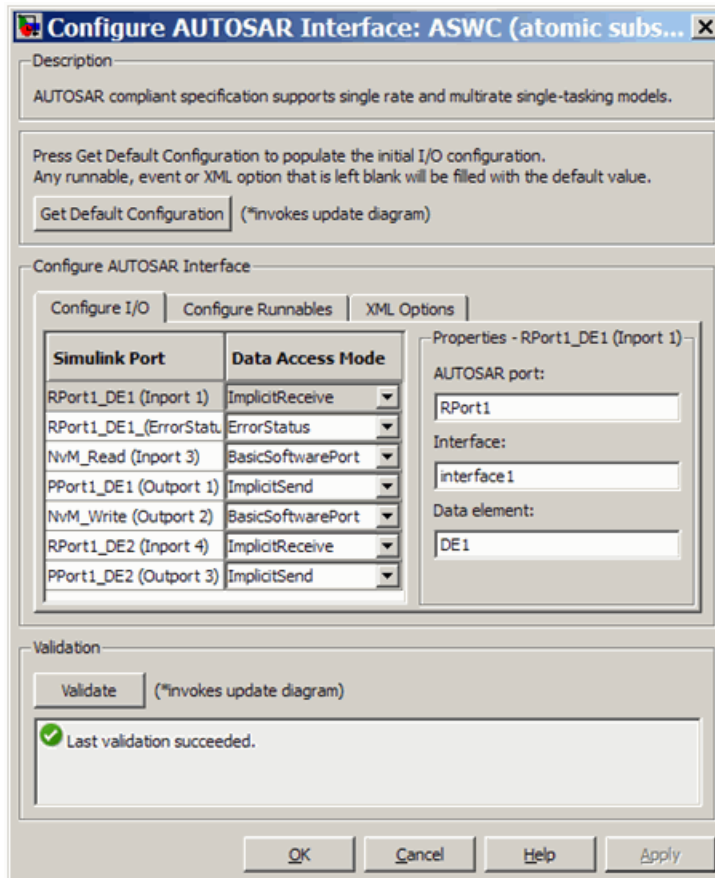


ASWC accesses data elements DE1 and DE2. You model data element access as follows:

- For Require ports, use Simulink inports. For example, RPort1_DE1 and RPort1_DE2.
- For Provide ports, use Simulink outports. For example, PPort1_DE1 and PPort1_DE2.

ErrorStatus is a value that the AUTOSAR Runtime Environment (RTE) returns to indicate errors that the communication system detects for each data element. You can use a Simulink inport to model error status, for example, RPort1_DE1 (ErrorStatus).

Use the Configure AUTOSAR Interface dialog box to specify the AUTOSAR settings for each inport and outport. The following figure shows settings for ASWC.

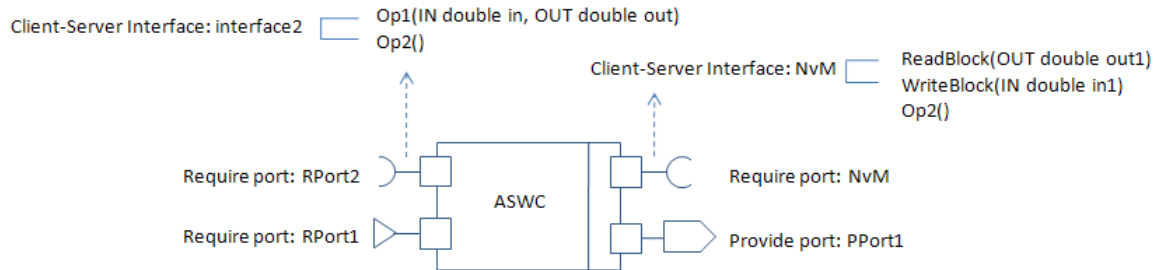


For example, the Data Access Mode for RPort1_DE1 is set to ImplicitReceive. For information on how you specify settings, see “Using the Configure AUTOSAR Interface Dialog Box” on page 22-32.

Client-Server Interface

A Client-Server Interface consists of one or more operation prototypes. An operation prototype contains one or more arguments of specific data types. A Client-Server Interface can be referenced by either a Require or Provide port.

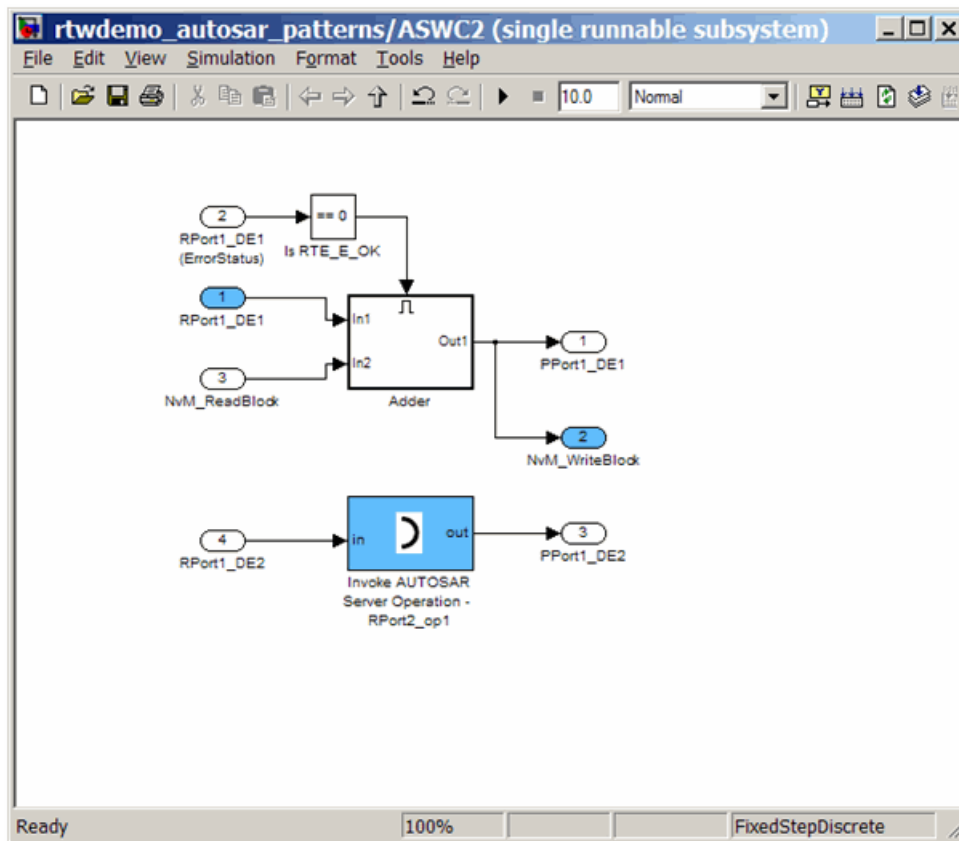
The following figure shows an AUTOSAR Software Component with Require ports (RPort2 and NvM) that reference Client-Server Interfaces (Interface2 and NvM).



Simulink provides the following modeling patterns for Client-Server Interfaces:

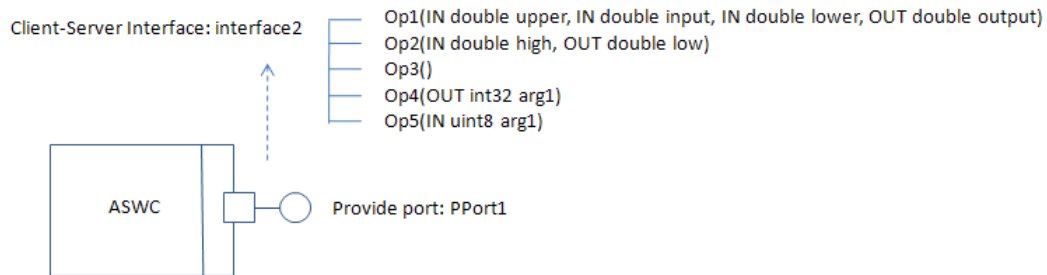
- If you want to invoke a Basic Software interface with operations that have only one argument, for example, Client-Server Interface: NvM, use an inport or outport.
- If you want to invoke Basic Software or application software interfaces that contain operations with any number of arguments, for example, Client-Server Interface: Interface2, use the Invoke AUTOSAR Server Operation block. See “Configuring the Invoke AUTOSAR Server Operation Block” on page 22-43

The following figure shows the use of the Invoke AUTOSAR Server Operation block in modeling an AUTOSAR Software Component in Simulink.

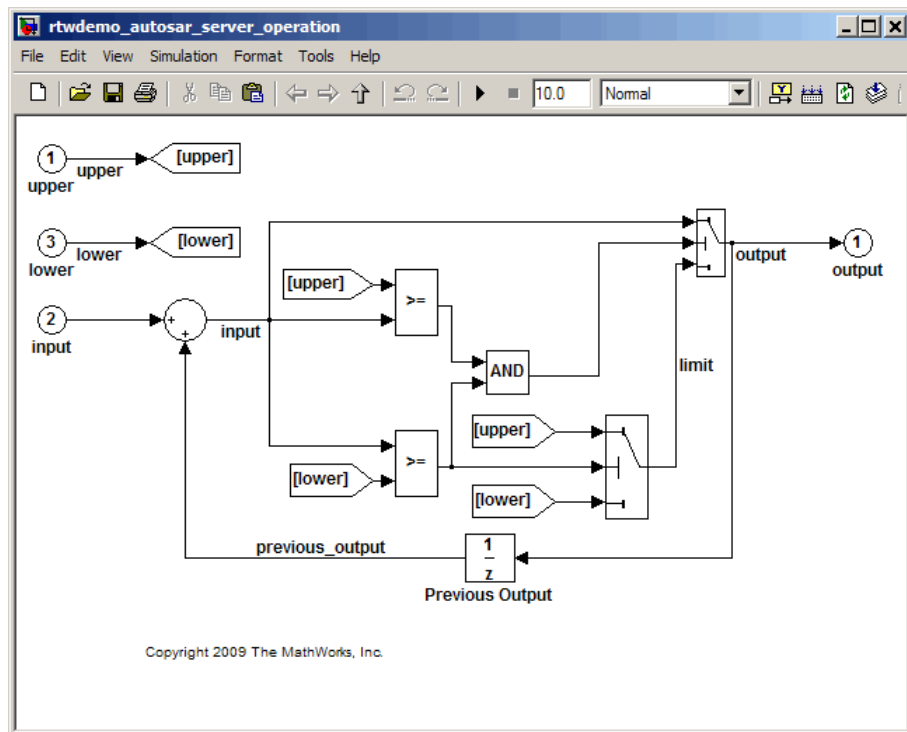


Use the Configure AUTOSAR Interface dialog box to specify the AUTOSAR settings for each inport and outport. See “Using the Configure AUTOSAR Interface Dialog Box” on page 22-32.

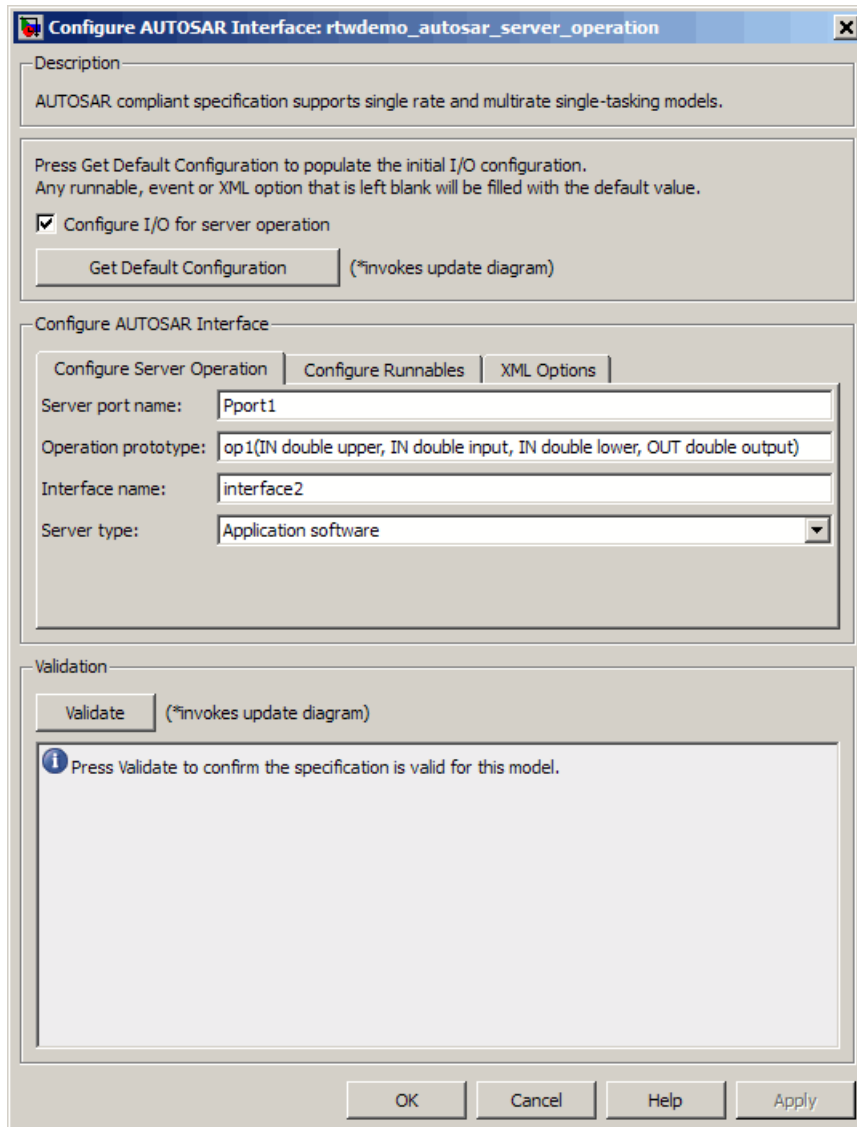
The following figure shows an AUTOSAR Software Component with a Provide port that references a Client-Server Interface.



In Simulink, you can model a single operation of an Atomic Software Component that is referenced by a Client-Server Interface. Consider the following model.



Use the Configure AUTOSAR Interface dialog box to map the inports and outputs to the arguments of the operation prototype. For example, the inports map to arguments upper, input, and lower.



For more information, see “Configuring a Server Operation” on page 22-40 .

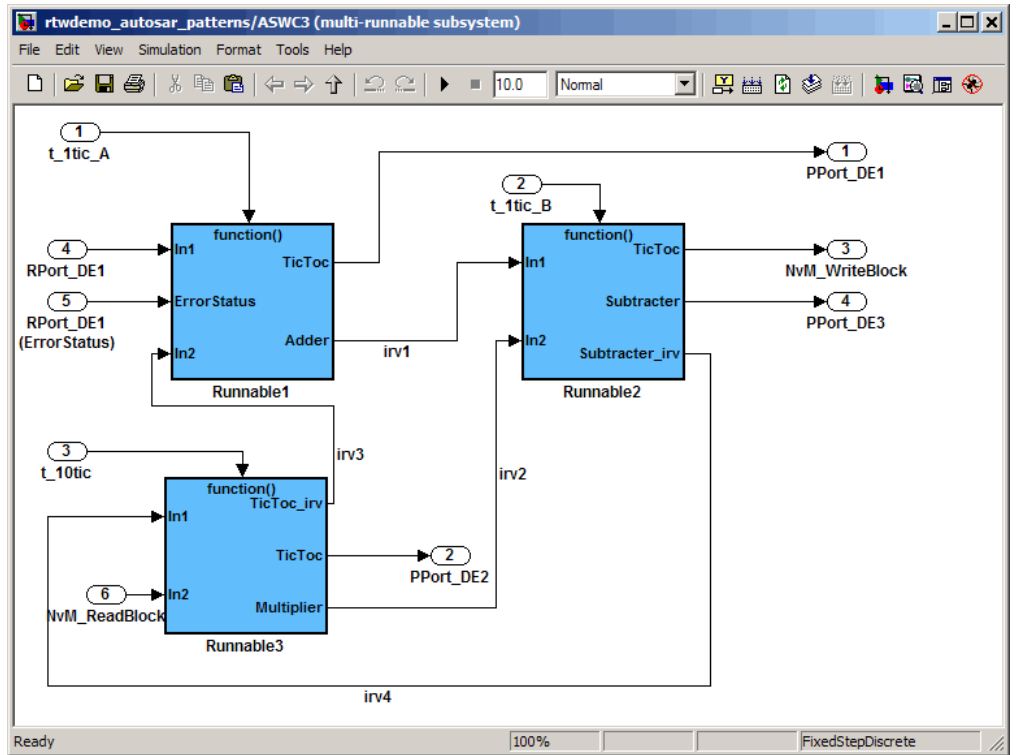
Calibration Parameters

AUTOSAR specifies a calibration component that contains calibration parameters used by AUTOSAR Software Components. You tune the calibration parameter values using calibration tools.

You can define the calibration component using an AUTOSAR authoring tool. You then import the calibration parameters into the MATLAB base workspace. You provide your Simulink model with access to these parameters by assigning the imported parameters to block parameters in your model. See “Importing an AUTOSAR Software Component” on page 22-30 and “Configuring Calibration Parameters” on page 22-48.

Inter-Runnable Variables

In AUTOSAR, *inter-runnable* variables are used to communicate primitive type data between runnables in the same component. You define these variables in a Simulink model by the signal lines that connect subsystems (runnables). For example, in the following figure, *irv1*, *irv2*, *irv3*, and *irv4* are inter-runnable variables.



See also “Configuring Multiple Runnables” on page 22-55.

Data Types

AUTOSAR specifies data types that apply to:

- Data elements of a Sender-Receiver Interface
- Operation arguments of a Client-Server Interface
- Calibration parameters
- Inter-runnable variables

The data types fall into two categories:

- Primitive data types, which allow a direct mapping to C intrinsic types.

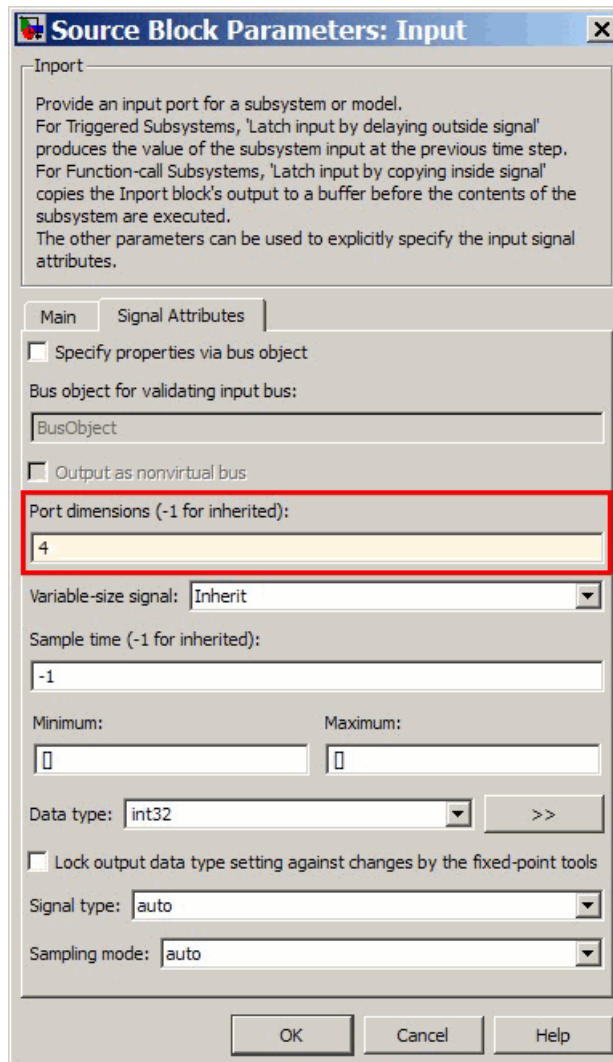
- Composite data types, which map to C arrays and structures.

You can use Simulink data types to define AUTOSAR primitive types.

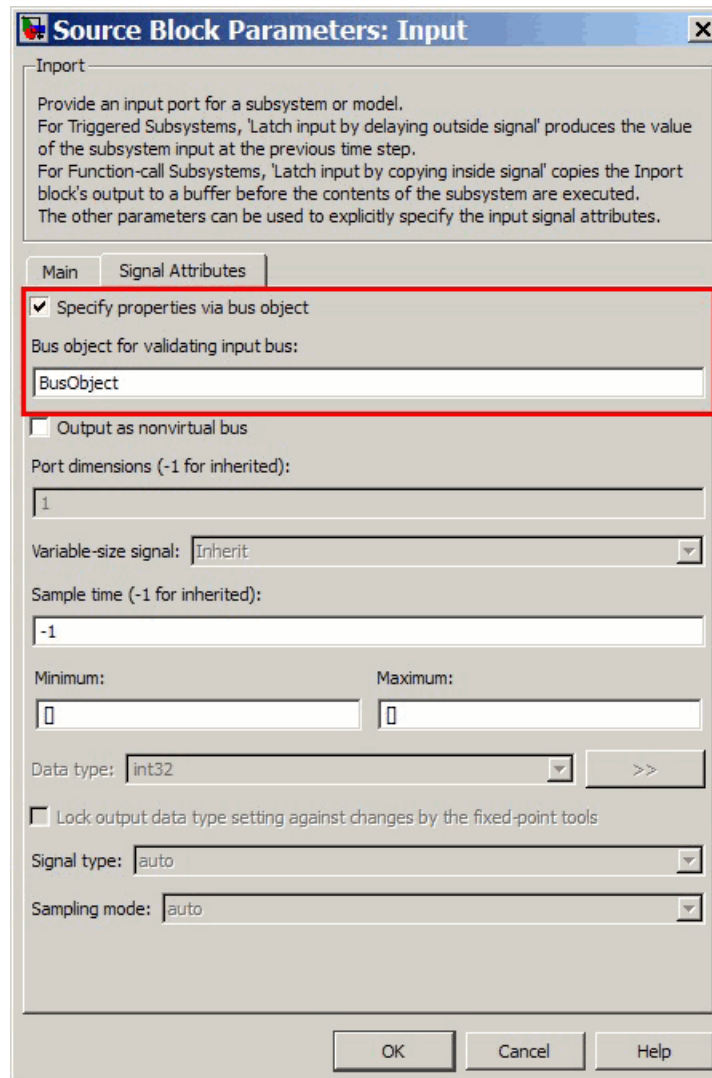
AUTOSAR Data Type	Simulink Data Type
UInt4	uint8
SInt4	int8
UInt8	uint8
SInt8	int8
UInt16	uint16
SInt16	int16
UInt32	uint32
SInt32	int32
Float_with_NaN	single
Float	single
Double_with_NaN	double
Double	double
Boolean	boolean
Char8	uint8
Char16	Not supported

AUTOSAR composite data types are arrays and records, which are represented in Simulink by wide signals and bus objects, respectively. In the Inport or Outport Block Parameters dialog box, use the **Signal Attributes** pane to configure wide signals and bus objects.

The following figure shows how to specify a wide signal, which corresponds to an AUTOSAR composite array.

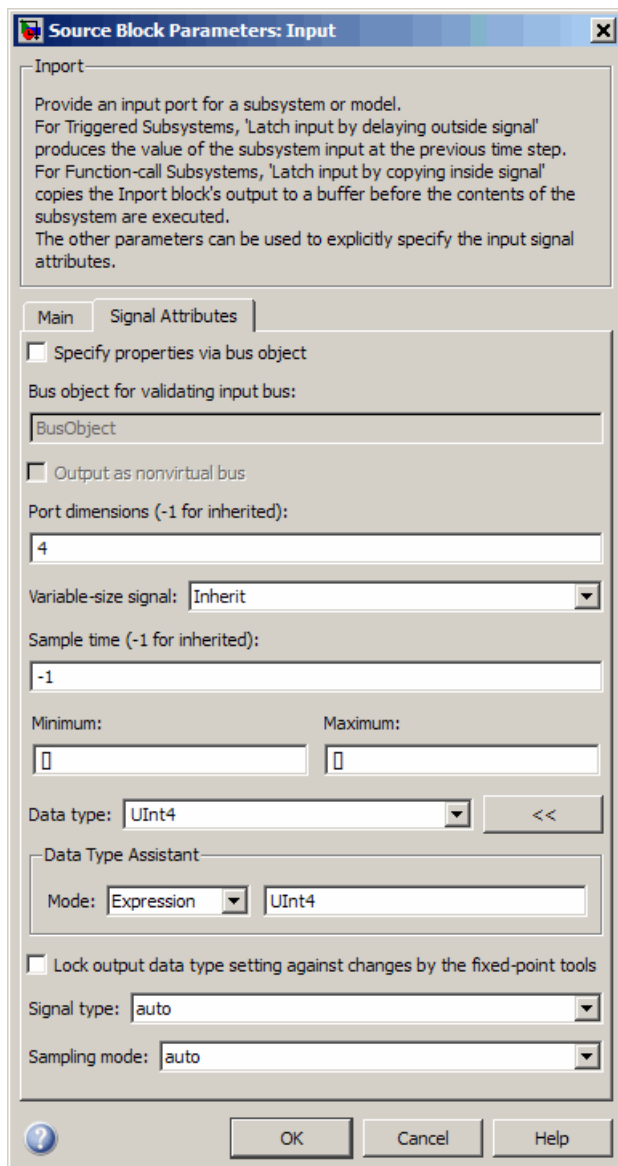


The following figure shows how to specify a bus object, which corresponds to an AUTOSAR composite record.



You can use the **Data Type Assistant** on the **Signal Attributes** pane of the Inport or Outport Block Parameters dialog box to specify the data types of data elements and arguments of an operation prototype. If you select **Mode** to be **Built in**, then you can specify the data type to be, for example, **single** or **boolean**. Alternatively, if you select **Mode** to be **Expression**, you can specify

an (alias) expression for data type. As an example, the following figure shows an alias UInt4 in the **Data type** field.



AUTOSAR Terminology

Term	Notes
AUTOSAR Runtime Environment (RTE)	<ul style="list-style-type: none"> • Layer between Application and Basic Software layers • Realizes communication between: <ul style="list-style-type: none"> ▪ AUTOSAR Software Components ▪ AUTOSAR Software Components and Basic Software
AUTOSAR Software Component	<ul style="list-style-type: none"> • A software component containing one or more algorithms, which communicates with its environment through ports • Connected to the AUTOSAR Runtime Environment (RTE) • Relocatable (not tied to a particular ECU)
Characteristics	Values of characteristics can be changed on an ECU through a calibration data management tool or an offline calibration tool.
Client-Server Interface	<ul style="list-style-type: none"> • PortInterface for client-server communication • Defines operations provided by server and used by client
Composite data types	Category of data types, such as one of the following: <ul style="list-style-type: none"> • Array — Contains more than one element of the same type, and has zero-based indexing • Record — Non-empty set of objects, where each object has a unique identifier
ComSpec	Defines specific communication attributes.
DataElementPrototype (data element)	Data value (signal) exchanged between a sender and a receiver.

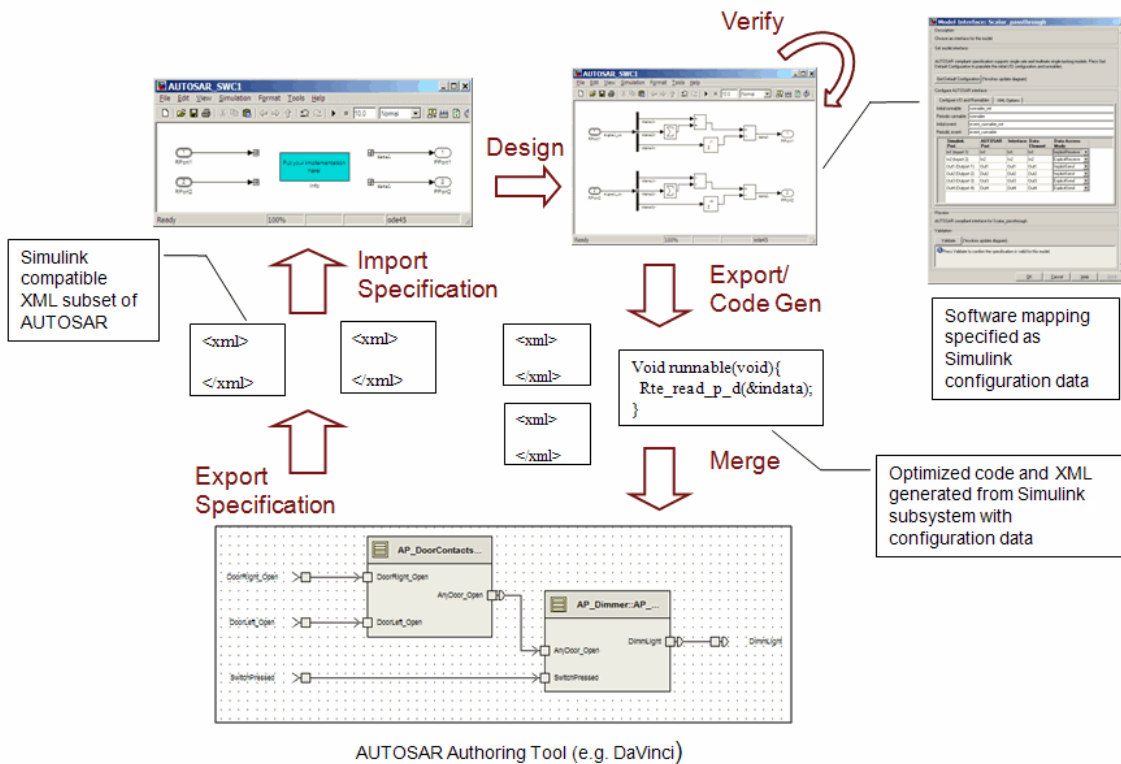
Term	Notes
Data types	<ul style="list-style-type: none"> • Either primitive or composite • Types data elements, arguments of operations in a Client-Server Interface, and constants
ErrorStatus	<p>Indicates errors detected by communication system. Runtime Environment defines the following macros for sender-receiver communication:</p> <ul style="list-style-type: none"> • RTE_E_OK: no errors • RTE_E_INVALID: data element invalid • RTE_E_MAX_AGE_EXCEEDED: data element outdated
OperationPrototype (operation)	<ul style="list-style-type: none"> • Invoked by a client • Provides value for each argument with direction in or inout, which must be of the correct data type • Client expects to receive a response to the invoked operation, part of which is a value (of correct data type) with direction out or inout
PortInterface	<ul style="list-style-type: none"> • Characterizes information provided or required by a port • Can be either Sender-Receiver Interface or Client-Server Interface
Primitive data types	Category of data types that allow a direct mapping to C intrinsic types.
Provide port (PPort)	Port providing data or service of a server.
Require port (RPort)	Port requiring data or service of a server.

Term	Notes
RTEEvent	<p>Event or situation that triggers execution of a runnable by the Runtime Environment (RTE). The software supports the following RTEEvents:</p> <ul style="list-style-type: none"> • OperationInvokedEvent (applicable to server operations) • TimingEvent
Runnable entity (runnable)	<p>Part of AUTOSAR Software-Component that can be executed and scheduled independently of other runnable entities (runnables).</p>
Sender-Receiver Interface	<ul style="list-style-type: none"> • PortInterface for sender-receiver communication • Defines data elements sent by sending component (with Provide port providing Sender-Receiver Interface) or received by receiving component (with Require requiring Sender-Receiver Interface)
Sender Receiver Annotation	<p>Annotation of data elements in a port that implements Sender-Receiver Interface.</p>
Sensor Actuator Software Component	<p>AUTOSAR Software Component dedicated to the control of a sensor or actuator.</p>
Service	<p>Logical entity of Basic Software that offers functionality, which is used by various AUTOSAR Software Components.</p>

Workflow for AUTOSAR

This section describes how you use Real-Time Workshop Embedded Coder software to generate AUTOSAR-compliant code.

The following diagram shows a workflow that you can follow.



You perform one or more of the following tasks:

- 1 Import previously specified AUTOSAR Software Components, including definitions of calibration parameters, into Simulink. See “Importing an AUTOSAR Software Component” on page 22-30 and “Configuring Calibration Parameters” on page 22-48.

- 2** Incorporate your Simulink design into the skeleton model or subsystem created by the import process.
- 3** Export, generating code and description files. This process involves configuring the AUTOSAR interface, validating this interface, and then building your Simulink models. See:
 - “Using the Configure AUTOSAR Interface Dialog Box” on page 22-32
 - “Configuring Ports for Basic Software and Error Status Receivers” on page 22-37
 - “Configuring Client-Server Communication” on page 22-39
 - “Configuring AUTOSAR Options Programmatically” on page 22-58
 - “Modifying and Validating an Existing AUTOSAR Interface” on page 22-49
 - “Exporting a Single Runnable AUTOSAR Software Component” on page 22-50
 - “Configuring Multiple Runnables” on page 22-55

You can also verify your generated code in a simulation. See “Verifying the AUTOSAR Code with SIL and PIL Simulations” on page 22-59.

- 4** Merge generated code and description files with other systems using an AUTOSAR authoring tool, for example, the DaVinci tool suite from Vector Informatik GmbH. See demo `rtwdemo_autosar_roundtrip_script`.

You can use the authoring tool to export specifications, which can be imported back into Simulink.

Importing an AUTOSAR Software Component

Use the `arxml.importer` class to parse an AUTOSAR Software Component description file (for example, exported from the DaVinci tool suite from Vector Informatik GmbH) and import into a Simulink model for configuration, code generation, and export to XML. For a complete list of methods, see “AUTOSAR” in the Real-Time Workshop Embedded Coder Function Reference documentation. You use functions in the following order:

- 1 Call `arxml.importer`, for example, `arxml.importer('mySoftwareComponentFile.arxml')` to create an importer object that looks for atomic software components in the specified “main” XML file. You can see reports in the Command Window describing identified atomic software components. You can have multiple components.

For example:

```
The file "mySoftwareComponentFile.arxml" contains:
1 Atomic-Software-Component-Type:
  '/ComponentType/complex_type_component'
3 CalPrm-Component-Type:
  '/ComponentType/MyCalibComp1'
  '/ComponentType/MyCalibComp2'
  '/ComponentType/MyCalibComp3'
```

Use `setFile` to change the main file and update the list of components.

Each software component requires an `arxml.importer` object. For each `arxml.importer` object, specify the file that contains the software component that you want.

- 2 Use the `setDependencies` method if you need to specify additional dependent XML files containing the information that completes the software component description (for example, data types, interfaces). You can specify a cell array of files or a single file.

Complete specifying dependencies only for components that you intend to import into Simulink.

3 To import a parsed atomic software component into a Simulink model, call one of the following methods. If you have not specified all dependencies for the components, you will see errors.

- `arxml.importer.createComponentAsSubsystem` — Creates and configures a Simulink subsystem skeleton corresponding to the specified atomic software component description.
- `arxml.importer.createComponentAsModel` — Creates and configures a Simulink model skeleton corresponding to the specified atomic software component description.

For example:

```
importer_obj.createComponentAsModel('/ComponentType/complex_type_component')
```

- `arxml.importer.createCalibrationComponentObjects` — Creates Simulink calibration objects corresponding to the specified AUTOSAR calibration component description.

For example:

```
[success] = createCalibrationComponentObjects(importer_obj,  
'CreateSimulinkObject', 'true')
```

See also the limitation, “Cannot Import Internal Behavior” on page 22-62.

After you import your software component into Simulink, you can modify the skeleton model or subsystem. To configure AUTOSAR code generation options and XML export options, see “Using the Configure AUTOSAR Interface Dialog Box” on page 22-32 or “Configuring AUTOSAR Options Programmatically” on page 22-58.

To see how to import, modify, and export AUTOSAR Software Components, see the Import and Export an AUTOSAR Software Component demo.

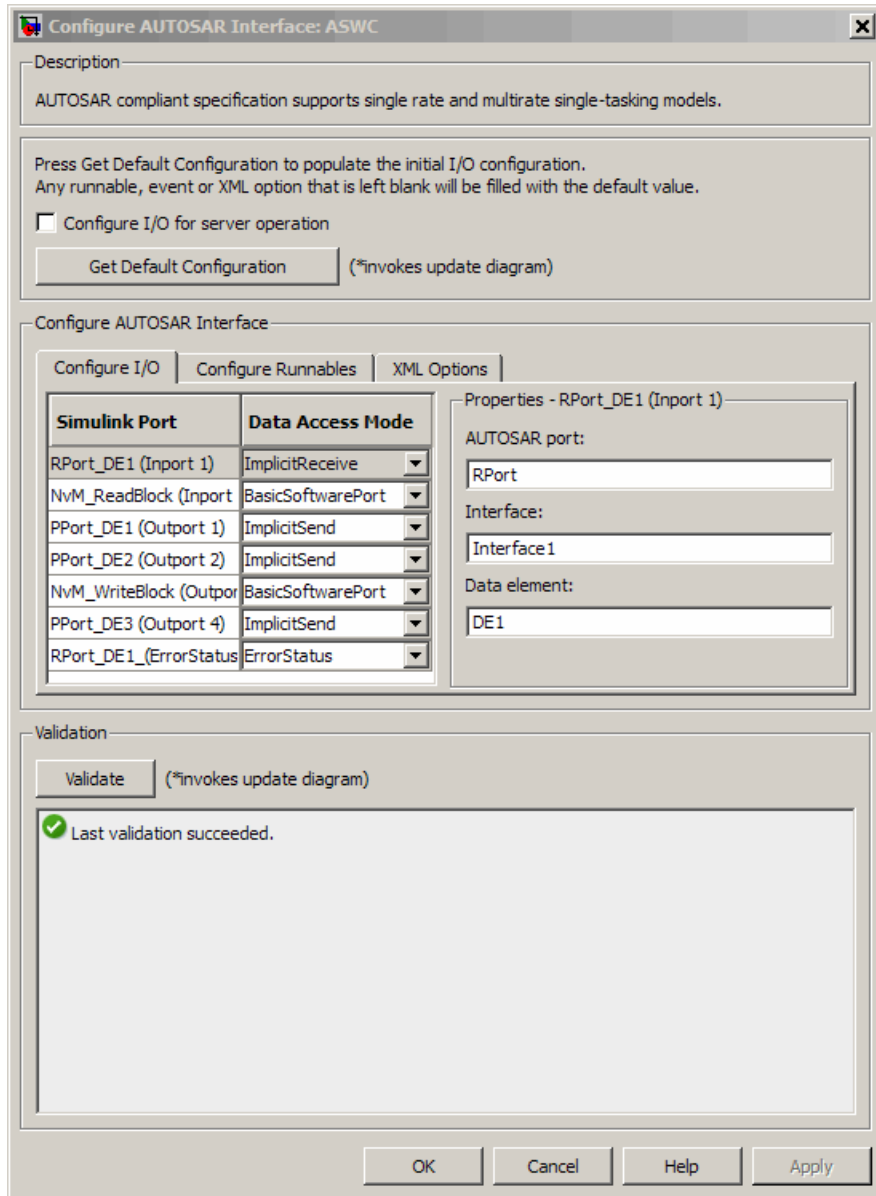
Using the Configure AUTOSAR Interface Dialog Box

Use the Configure AUTOSAR Interface dialog box to configure your AUTOSAR code generation and XML import and export options. Alternatively, you can use functions to control all AUTOSAR options programmatically.

In any model using the `autosar.tlc` system target file, you can open the Configure AUTOSAR Interface dialog box by right-clicking a subsystem and selecting **Real-Time Workshop > AUTOSAR *Single or Multi-Runnable Component* > Configure**.

Single-Runnable menu options are enabled only for atomic or function-call subsystems.

Multi-Runnable menu options are enabled only for virtual subsystems.

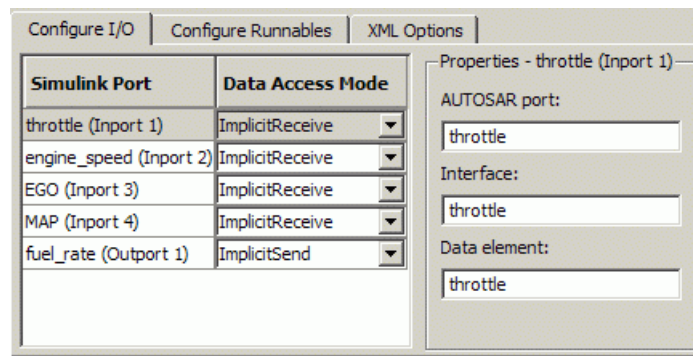


To configure your AUTOSAR options:

- 1 Clear the **Configure I/O for server operation** check box if it is selected. You select this check box only when you want to configure your Simulink model as a server operation (see “Configuring a Server Operation” on page 22-40).
- 2 Click **Get Default Configuration** to populate the controls for your model.

The runnable names, XML properties, and I/O configuration are initialized. If you click **Get Default Configuration** again later, only the I/O configurations are reset to default values.

- 3 In the **Configure AUTOSAR Interface** pane, use the controls to change your AUTOSAR code generation options and XML export options, for example, send and receive communication options such as port and interface names, data access modes, runnable, initialization, and periodic function names.
 - On the **Configure I/O** tab, designate inports and outports as data sender/receiver ports, error status receivers, or as access points to basic software.



To designate inports and outports as sender or receiver ports, set each port’s **Data Access Mode** to one of the following:

- **Implicit** (recommended). Data is buffered by the Runtime Environment (RTE).
- **Explicit**. Data is not buffered.

Use the port interface settings to reflect your AUTOSAR port best practices. For example, some AUTOSAR users like to group related

data into the same AUTOSAR port. You can achieve this arrangement in the GUI by duplicating AUTOSAR port names. Alternatively, you can use the AUTOSAR port to group information individually; in this case, a common approach is to set all of the data element settings to something neutral, for example, 'data', and leave the AUTOSAR port names as they are. You can also use the AUTOSAR interface name for any best practices that you might have. For example, you can set up interfaces for individual AUTOSAR ports by ensuring that the interface names change when the AUTOSAR port name changes, for example, by prefixing the AUTOSAR interface of the corresponding AUTOSAR port name with an 'if_'.

For more information on all these options, see “AUTOSAR” in the Real-Time Workshop Embedded Coder Function Reference documentation.

You also use **Data Access Mode** to designate ports to access basic software or error status. See “Configuring Ports for Basic Software and Error Status Receivers” on page 22-37.

- On the **Configure Runnables** tab, specify the names of your runnable, initialization, and periodic functions.

Configure I/O	Configure Runnables	XML Options
Initial runnable:	Runnable_fuel_Init	
Initial event:	Event_fuel_Init	
Periodic runnable:	Runnable_fuel_Step	
Periodic event:	Event_fuel_Step	

- On the **XML Options** tab, specify the names and package paths of the XML files that you publish when you generate code. For more details about these files, see “Exporting a Single Runnable AUTOSAR Software Component” on page 22-50.

Configure I/O	Configure Runnables	XML Options	
Interface package:	/fuel_pkg/fuel_if		
Component name:	/fuel_pkg/fuel_swc/fuel		
Internal behavior name:	/fuel_pkg/fuel_ib/fuel		
Implementation name:	/fuel_pkg/fuel_imp/fuel		
Data type package:	/fuel_pkg/fuel_dt		

- 4 After you configure your options, click **Validate**, which calls `runValidation`. If there are problems, you see messages describing why the configuration is invalid.

Note For information on all validation checks, see `RTW.AutosarInterface.runValidation` in the Real-Time Workshop Embedded Coder Function Reference documentation.

- 5 If validation succeeds, click **OK** to return to the Configuration Parameters dialog box.
- 6 Save your model and then generate code to export your AUTOSAR component.

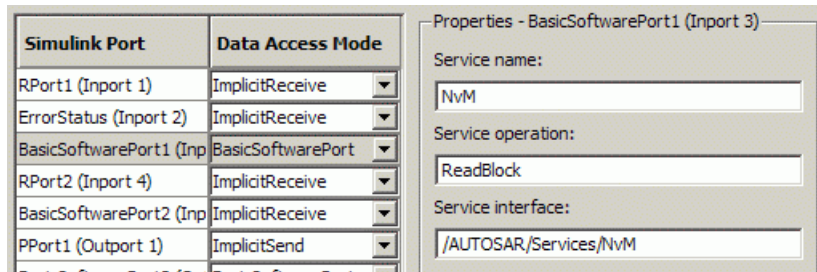
Configuring Ports for Basic Software and Error Status Receivers

You can configure ports to access AUTOSAR services and device drivers (AUTOSAR basic software), and to access communication error status in your model. You can configure ports programmatically or by using the AUTOSAR Model Interface dialog box. To open the dialog box, right-click a subsystem and select **Real-Time Workshop > AUTOSAR Single or Multirunnable Component > Configure**.

In the dialog box, you can specify the **Data Access Mode** of every port.

- Designate inports and outports as access points to basic software.

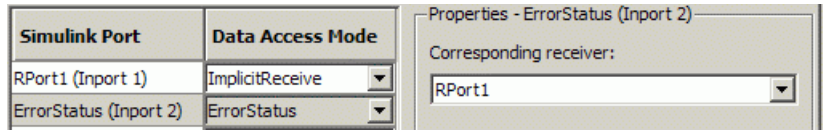
If you select **Basic Software**, specify the service name, operation, and interface. The service name and operation must be valid AUTOSAR identifiers, and the service interface must be a valid path of the form `AUTOSAR/Service/servicename`.



After you export your AUTOSAR components, you must include your service interface definition XML file to import correctly into an authoring tool.

- Designate inports to receive error status.

If you select **Error Status** for an inport, you must select the other port (of mode Implicit or Explicit Receive) to listen to, for error status. Error status ports must use `uint8` data type (or an alias).



Configuring Client-Server Communication

In this section...

“Configuring a Server Operation” on page 22-40

“Configuring the Invoke AUTOSAR Server Operation Block” on page 22-43

“Creating Configurable Subsystems from a Client-Server Interface” on page 22-45

“Simulating and Generating Code for Client-Server Communication” on page 22-46

AUTOSAR allows client-server communication between:

- Application software components
- An application software component and Basic Software

An AUTOSAR Client-Server Interface defines the interaction between a software component that *provides* the interface and a software component that *requires* the interface. The component that provides the interface is the server. The component that requires the interface is the client.

In Simulink, you can:

- Configure your model to implement a server operation. You generate AUTOSAR-compliant code and XML description files, including a client-server interface, when you build your model. See “Configuring a Server Operation” on page 22-40.
- Configure a client port for your model using an Invoke AUTOSAR Server Operation block that references a client-server interface. You generate AUTOSAR-compliant code and XML description files for your client port when you build your model. See “Configuring the Invoke AUTOSAR Server Operation Block” on page 22-43.

Once you create a client-server interface, you can generate a Simulink library of configurable, client-server subsystems that reference the:

- Invoke AUTOSAR Server Operation block for code generation

- Server operation model block for simulation

For information on how to generate this library, see “Creating Configurable Subsystems from a Client-Server Interface” on page 22-45

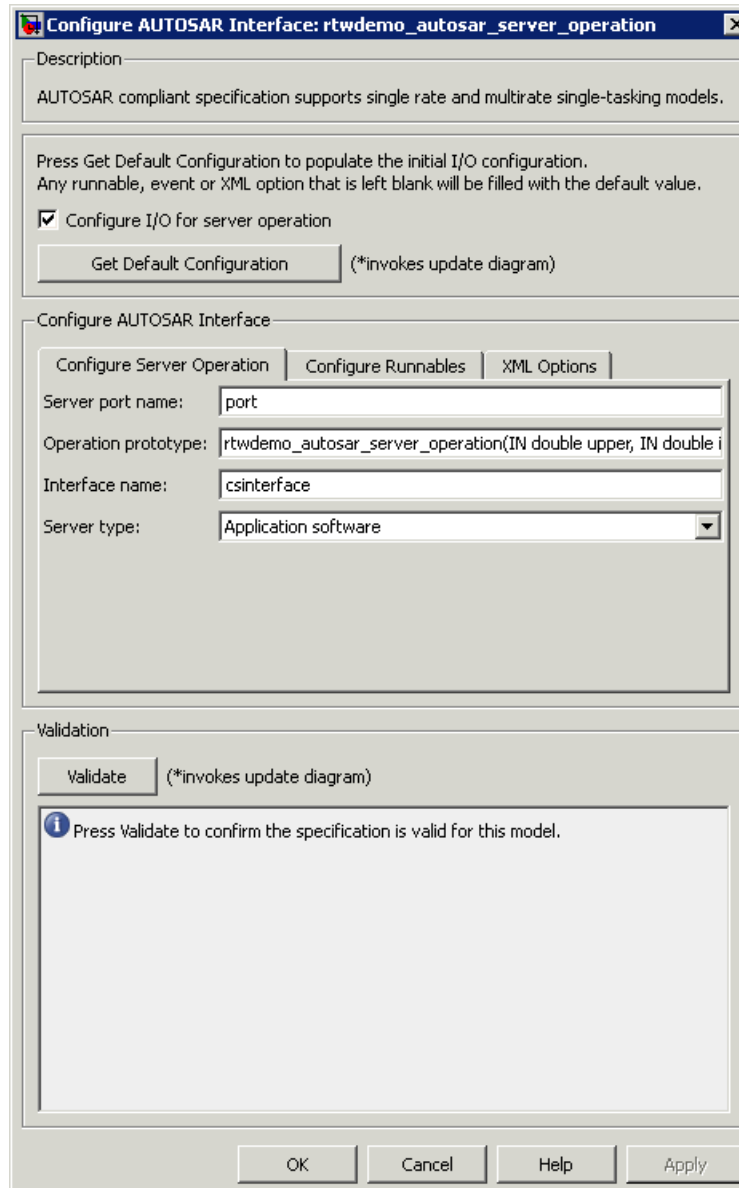
You can deploy the client-server subsystem in a Simulink model and, using the Mode Switch for Invoke AUTOSAR Server Operation, run the model in either a simulation or code-generation mode. See “Simulating and Generating Code for Client-Server Communication” on page 22-46.

For a demo on generating and using an AUTOSAR Client-Server Interface, see `rtwdemo_autosar_clientserver_script`.

Configuring a Server Operation

In the Configure AUTOSAR Interface dialog box, you can configure your Simulink model as a server operation and then generate AUTOSAR-compliant code and XML files, including the client-server interface.

- 1 Select the **Configure I/O for server operation** check box. The **Configure I/O** tab becomes the **Configure I/O for server operation** tab.



2 Click **Get Default Configuration** to populate the controls for your model.

The runnable names, XML properties, and I/O configuration are initialized. If you click **Get Default Configuration** again later, only the I/O configurations are reset to default values.

Use the controls in the **Configure AUTOSAR Interface** pane to change your AUTOSAR code generation options and XML export options.

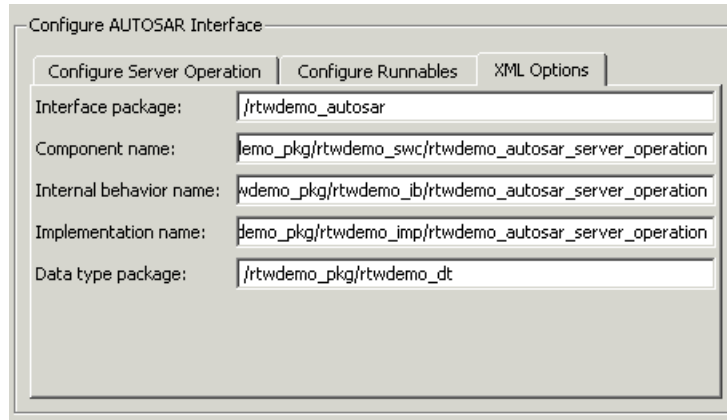
- 3 On the **Configure Server Operation** tab, specify the following:
 - **Server port name.** Use a valid AUTOSAR short-name identifier.
 - **Operation prototype .** The names of the prototype and its arguments must be valid AUTOSAR short-name identifiers, for example `rtwdemo_autosar_server_operation(IN double upper, IN double input, IN double lower, OUT double output)`.
 - **Interface name.** The path reference of the client-server interface. Use a valid AUTOSAR short-name path, for example, `csinterface`
 - **Server type.** From the drop-down list, select either `Application software` or `Basic software`.
- 4 On the **Configure Runnables** tab, specify the names of your runnable, initialization, and periodic functions.

The screenshot shows a dialog box titled "Configure AUTOSAR Interface". It has three tabs: "Configure Server Operation", "Configure Runnables", and "XML Options". The "Configure Runnables" tab is selected. Inside this tab, there are four text input fields arranged vertically:

- Initial runnable:
- Initial event:
- Periodic runnable:
- Periodic event:

- 5 On the **XML Options** tab, specify the names and package paths of the XML files that you publish when you generate code. For more details

about these files, see “Exporting a Single Runnable AUTOSAR Software Component” on page 22-50.



- 6 After you configure your options, click **Validate**, which calls `runValidation`. If there are problems, you see messages describing why the configuration is invalid.

Note For information on all validation checks, see `RTW.AutosarInterface.runValidation` in the Real-Time Workshop Embedded Coder Function Reference documentation.

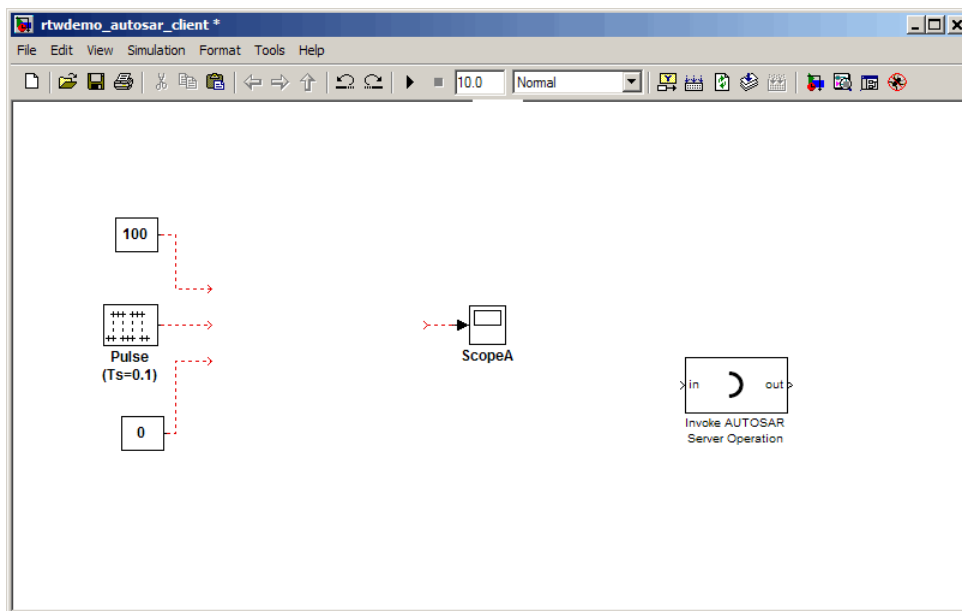
- 7 If validation succeeds, click **OK** to return to the Configuration Parameters dialog box.
- 8 Save your model.
- 9 To generate AUTOSAR-compliant code and XML files, select **Tools > Real-Time Workshop > Build Model**.

Configuring the Invoke AUTOSAR Server Operation Block

You can use the Invoke AUTOSAR Server Operation block in your Simulink model to configure a client port (that accesses either application software

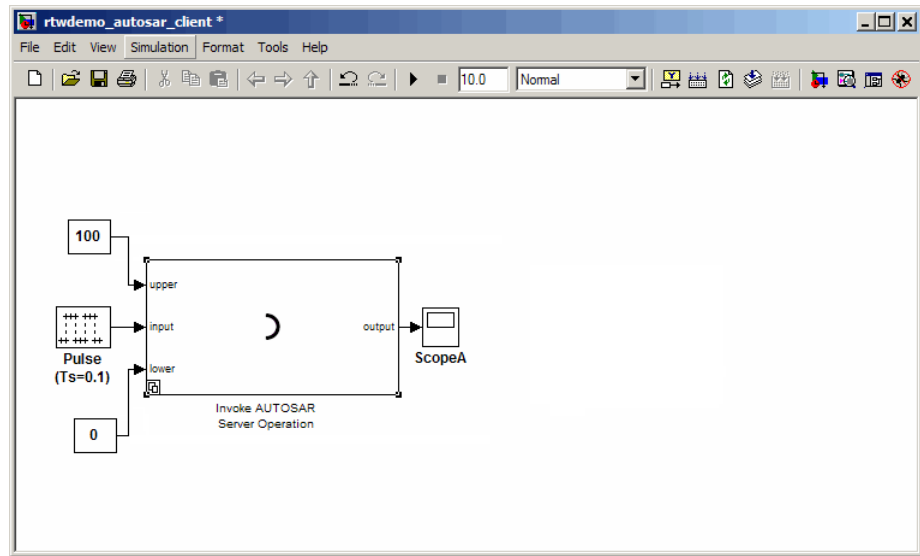
or AUTOSAR Basic Software). You can then build the model to generate AUTOSAR-compliant code and XML files.

- 1 Drag an Invoke AUTOSAR Server Operation block into your model.



- 2 Double-click the block to open the Invoke AUTOSAR Server Operation dialog box. Specify the following:
 - **Client port name.** A valid AUTOSAR short-name identifier
 - **Operation prototype.** The names of the prototype and its arguments must be valid AUTOSAR short-name identifiers, for example, `rtwdemo_autosar_server_operation(IN double upper, IN double input, IN double lower, OUT double output)`
 - **Interface path.** The path reference of the client-server interface. You must use a valid AUTOSAR short-name path, for example, `/AUTOSAR/Interface`.
 - **Server type.** From the drop-down list, select either Application software or Basic software.

- **Show error status.** Select this check box if you want the client port to receive the error status of client-server communication.
 - **Sample time.** Set this parameter to -1 to inherit the sample time.
- 3 Click **OK**. Your Invoke AUTOSAR Server Operation block is updated.
 - 4 Connect the updated Invoke AUTOSAR Server Operation block to your model.



- 5 Select **Tools > Real-Time Workshop > Build Model**.
AUTOSAR-compliant code and XML files for the client port are generated.

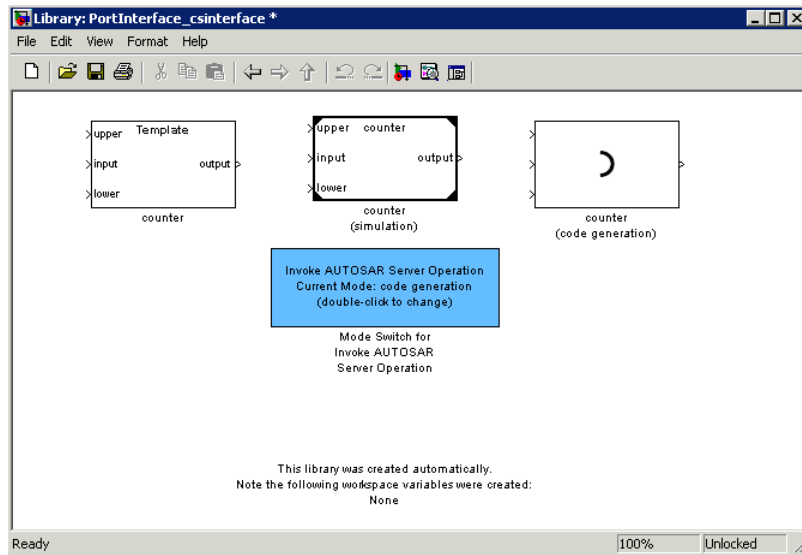
Creating Configurable Subsystems from a Client-Server Interface

You can generate a Simulink library of configurable subsystems by applying the `createOperationAsConfigurableSubsystems` method to the `arxml.importer` object with the client-server interface. For example:

```
% Create an AUTOSAR importer object
obj = arxml.importer('rtwdemo_autosar_csinterface.arxml');
```

```
% Create the client-server operation configurable subsystem library
obj.createOperationAsConfigurableSubsystems('/PortInterface/csinterface', ...
                                           'CreateSimulinkObject', false);
```

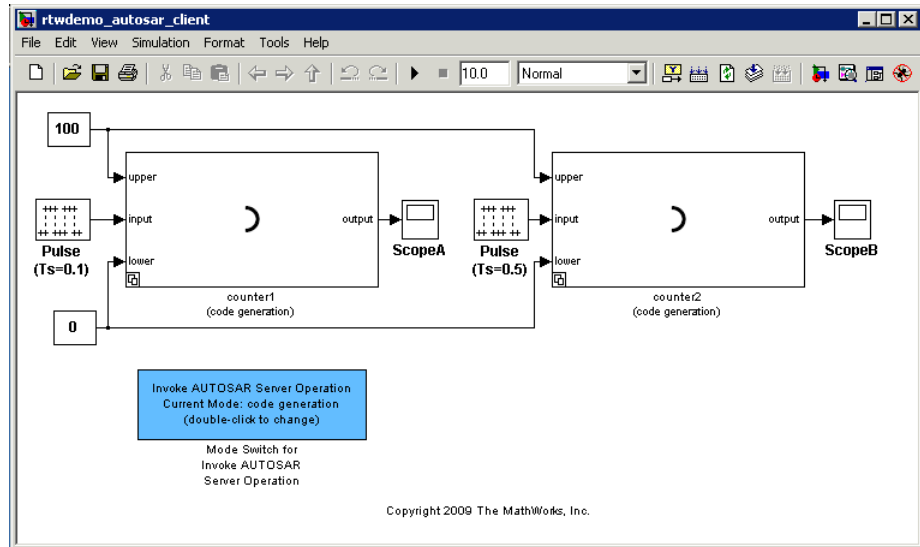
yield the following PortInterface_csinterface library.



Simulating and Generating Code for Client-Server Communication

Use the Template block from the client-server subsystem library to construct a model that can be run in either code-generation or simulation mode.

- 1 Drag the Template block from the subsystem library into your model window and connect it to other blocks.
- 2 Place the Mode Switch for Invoke AUTOSAR Server Operation in your model window.



To simulate the model:

- 1** Double-click the Mode Switch for AUTOSAR Server Operation to change the current mode from code generation to simulation.
- 2** Select **Simulation > Start**.

To generate code for the model:

- 1** Double-click the Mode Switch for AUTOSAR Server Operation to change the current mode from simulation to code generation.
- 2** Select **Tools > Real-Time Workshop > Build Model**.

Configuring Calibration Parameters

To import calibration parameters, use the importer method `arxml.importer.createCalibrationComponentObjects`. This method imports all of your parameters into the MATLAB workspace, and you can then assign them to block parameters in your model.

So that the calibration parameters export correctly when you generate code, check the following configuration parameter settings:

- **Inline parameters** must be selected
- **Custom Storage Class not ignored** must not be selected

After you export your AUTOSAR components, you must include your calibration interface definition XML file to import correctly into an authoring tool.

Modifying and Validating an Existing AUTOSAR Interface

You can validate your AUTOSAR interface using the Configure AUTOSAR Interface dialog box. See “Using the Configure AUTOSAR Interface Dialog Box” on page 22-32. The following steps show how you can modify and validate your AUTOSAR interface *programmatically*:

- 1 Get the handle to an existing model-specific `RTW.AutosarInterface` object that is attached to your loaded Simulink model. Enter:

```
obj = RTW.getFunctionSpecification(modelName)
```

`modelName` is a string specifying the name of a loaded Simulink model, and `obj` returns a handle to an `RTW.AutosarInterface` object attached to the specified model.

Test the AUTOSAR interface object. Enter:

```
isa(obj, 'RTW.AutosarInterface')
```

This test must return 1. If the model does not have an AUTOSAR interface object, the function returns [].

- 2 To view and change items, use the AUTOSAR `get` and `set` functions listed in “AUTOSAR” in the Real-Time Workshop Embedded Coder Function Reference documentation.
- 3 Validate the function prototype using `RTW.AutosarInterface.runValidation`.

Note For information on all validation checks, see `RTW.AutosarInterface.runValidation` in the Real-Time Workshop Embedded Coder Function Reference documentation.

- 4 If validation succeeds, save your model and then generate code.

Exporting a Single Runnable AUTOSAR Software Component

After configuring your AUTOSAR export options, generate code to export your AUTOSAR Software Component.

Building the subsystem or model generates the code and XML files according to your customizations.

The software component C code and the following XML files are exported to the build folder.

File Name	Description
<i>modelName_behavior.xml</i>	Specifies the software component internal behavior.
<i>modelName_implementation.xml</i>	Specifies the software component implementation.
<i>modelName_interface.xml</i>	Specifies the software component interfaces, including extra interfaces.
<i>modelName_component.xml</i>	Specifies the software component type, including additional ports added to the Simulink model.
<i>modelName_datatype.xml</i>	Specifies the software component data types, including any modified or additional data types.

Note In addition to the AUTOSAR software component C code, Real-Time Workshop Embedded Coder creates the following header files in the `stub` subfolder of the build folder:

- `Rte_Type.h`
- `Rte_SWC.h`, where *SWC* is the name of the software component
- `Compiler.h`

These files contain dummy implementations of AUTOSAR functions, which the software uses for SIL and PIL simulations. You must *not* use these files outside Simulink. Your AUTOSAR RTE generator should produce the equivalent files.

You can merge the software component information back into an AUTOSAR authoring tool. This software component information is partitioned into separate files to facilitate merging. The partitioning attempts to minimize the number of merges you need to do. In general, you do not need to merge the data type file into the authoring tool because data types are usually defined early in the design process. You must, however, merge the internal behavior file because this information is part of the model implementation.

For an example of how to generate AUTOSAR-compliant code and export AUTOSAR Software Component description XML files from a Simulink model, see the AUTOSAR Code Generation demo.

Selecting an AUTOSAR Schema

The default AUTOSAR schema version is 3.0. If you need to change the schema version, you must do so before exporting.

To select a schema version, open the Configuration Parameters dialog box:

- 1 In any model using the `autosar.tlc` system target file, the **AUTOSAR Code Generation Options** component appears in the tree.

Click **AUTOSAR Code Generation Options** to open the **AUTOSAR Code Generation Options** pane.

2 Select a schema version (3.0, 2.1 or 2.0) for generating XML files.

Tip While you are working in this view, you can also click the **Configure AUTOSAR Interface** button to open the Configure AUTOSAR Interface dialog box.

Generating AUTOSAR Compiler Abstraction Macros

Compilers for 16-bit platforms (for example, Cosmic and Metrowerks for S12X or Tasking for ST10) use special keywords to deal with the limited 16-bit addressing range. The location of data and code beyond the 64 k border is selected explicitly by special keywords. However, if such keywords are used directly within the source code, then software must be ported separately for each microcontroller family, that is, the software is not platform-independent.

AUTOSAR specifies C macros to abstract compiler directives (near/far memory calls) in a platform-independent manner. These compiler directives, derived from the 16-bit platforms, enable better code efficiencies for 16-bit micro-controllers without separate porting of source code for each compiler. This approach allows your system integrator, rather than your software component implementer, to choose the location of data and code for each software component.

For more information on AUTOSAR compiler abstraction, see www.autosar.org

Configuring AUTOSAR Compiler Macro Generation

Before you build your model, on the Real-Time Workshop AUTOSAR Code Generation Options pane, select the **Use AUTOSAR compiler abstraction macros** check box.

When you build the model, the software applies compiler abstraction macros to global data and function definitions in the generated code.

For data, the macros are in the following form:

- `CONST(consttype, memclass) varname;`

- `VAR(type, memclass) varname;`

where

- *consttype* and *type* are data types
- *memclass* is a macro string `SWC_VAR` (*SWC* is the software component identifier)
- *varname* is the variable identifier

For functions (model and subsystem), the macros are in the following form:

- `FUNC(type, memclass) funcname(void)`

where

- *type* is the data type of the return argument
- *memclass* is a macro string. This string can be either `SWC_CODE` for runnables (external functions), or `SWC_CODE_LOCAL` for internal functions (*SWC* is the software component identifier).

Example

If the **Use AUTOSAR compiler abstraction macros** check box is *not* selected, the software generates the following code:

```
/* Block signals (auto storage) */
BlockIO rtB;

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

/* Model step function */
void Runnable_Step(void)
```

However, if the **Use AUTOSAR compiler abstraction macros** check box is selected, the software generates macros in the code:

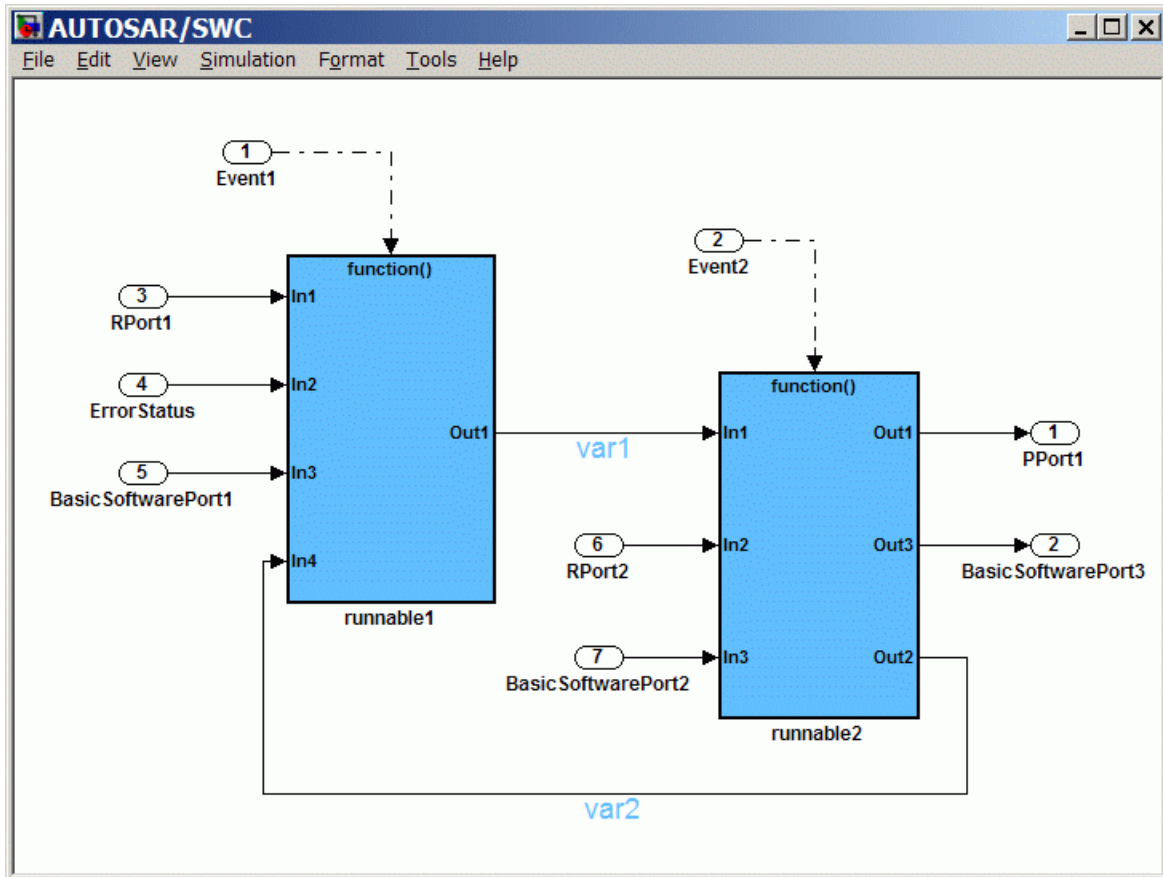
```
/* Block signals (auto storage) */
VAR(BlockIO, SWC1_VAR) rtB;
```

```
/* Block states (auto storage) */  
VAR(D_Work, SWC1_VAR) rtDWork;  
  
/* Model step function */  
FUNC(void, SWC1_CODE) Runnable_Step(void)
```

Configuring Multiple Runnables

You can use subsystems to model multiple runnables explicitly in a single AUTOSAR Software Component. You can map AUTOSAR runnables to function-call subsystems by using the Export Functions feature. If your model contains function-call subsystems, as in the following example, you can use the Export Functions command to specify that you want each subsystem to represent an AUTOSAR runnable.

Specify an AUTOSAR interface for each function call subsystem being exported as a runnable. Configure the AUTOSAR interfaces for your subsystems by right-clicking the top-level wrapper subsystem and selecting **Real-Time Workshop > AUTOSAR Multi-Runnable Component > Configure**.



To specify that you want multiple runnables generated from your subsystems, right-click the top-level subsystem and select **Real-Time Workshop > AUTOSAR Multi-Runnable Component > Export Functions**.

This command builds code for an AUTOSAR runnable for each subsystem. In the example shown, `runnable1` and `runnable2` are both runnables. The build also creates an additional runnable at code generation time to aggregate the initialization functions for each of the function-call subsystems.

Inter-runnable variables communicate data between runnables to ensure data integrity. Define these inter-runnable variables by the signal lines connecting subsystems (var1 and var2 in the example). Double-click to edit these signal names before generating code.

In a multi-runnable software component, you may need to use blocks that depend on time, such as the Discrete-Time Integrator block. You can use a timer for each AUTOSAR runnable if a block uses time. The timer increments at each execution of the runnable. You specify the timer resolution in the AUTOSAR Model Interface dialog box, in the **Execution period** field. In the Configuration Parameters dialog box, under Optimization, specify the **Application life span (days)**. The timer data type is based on your specified application life span and execution period.

To see how to configure and generate AUTOSAR-compliant code and export AUTOSAR Software Component description XML files for a Simulink model with multiple runnables, see the AUTOSAR Code Generation for Multiple Runnable Entities demo.

Configuring AUTOSAR Options Programmatically

To control AUTOSAR options programmatically, use the AUTOSAR functions listed in the following tables in the Real-Time Workshop Embedded Coder Function Reference documentation.

- “AUTOSAR Component Import”
- “AUTOSAR Configuration”

Verifying the AUTOSAR Code with SIL and PIL Simulations

In this section...

“Overview” on page 22-59

“Using the SIL and PIL Simulation Modes” on page 22-59

“Using a SIL Block for AUTOSAR Verification” on page 22-60

Overview

You can carry out model-based verification of AUTOSAR software components using software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations. Use SIL for verification of generated source code on your host computer, and PIL for verification of object code on your production target.

Using the SIL and PIL Simulation Modes

You can run a top model or Model block that is configured for the AUTOSAR target (`autosar.tlc`) using the Software-in-the-Loop (SIL) and Processor-in-the-Loop (PIL) simulation modes.

For more information:

- Top-model and Model block SIL simulation, see “Configuring a Top-Model SIL Simulation” on page 36-18 and “Configuring a Model Block SIL Simulation” on page 36-20.
- Top-model and Model block PIL simulation, see “Configuring a PIL Simulation” on page 39-14.

AUTOSAR Top Model SIL and PIL Support

For the SIL and PIL simulation modes, the software supports the following AUTOSAR features:

- Single runnable atomic software components
- Sender-receiver ports
- Basic Software ports

- Error status

For a top model running in SIL or PIL simulation mode, the software does *not* support the following AUTOSAR features:

- AUTOSAR calibration parameters
- Client-server operations
- Multirunnables (export functions)

Logging Invariant Output Signals. Through signal logging, you can configure your top model to log invariant output signals. However, the software will log these invariant signals as periodically sampled data.

AUTOSAR Model Block SIL and PIL Support

The software supports testing of AUTOSAR components that are modeled as model reference components. These model reference components are implemented as standard model reference Real-Time Workshop targets and do not contain any special AUTOSAR behavior.

Using a SIL Block for AUTOSAR Verification

You can create a SIL block, which wraps the generated code in an S-function. The AUTOSAR target automatically configures the generated S-function to route simulation data using AUTOSAR run-time environment (RTE) API calls.

To verify generated code using a SIL block:

- 1** In the Configuration Parameters dialog box, select **Real-Time Workshop > SIL and PIL Verification > Create SIL block**. Alternatively, in the Command Window, you can run the following commands:

```
>> set_param( modelName, 'GenerateErtSFunction', 'on' );  
>> set_param( modelName, 'GenCodeOnly', 'off' );
```

:

- 2** Generate code to build the SIL block.

- 3** Once the SIL block is built, replace the existing component in your model with the new block.
- 4** Simulate the model and check the output to verify that the code produces the same data as the original subsystem.

See also “Verifying Code Using a SIL Block” on page 36-30.

Limitations and Tips

In this section...
“Cannot Import Internal Behavior” on page 22-62
“Cannot Copy Subsystem Blocks Without Losing Interface Information” on page 22-62
“Error If No Default Configuration” on page 22-63
“Server Operation Model with Reusable Function Subsystems” on page 22-63
“Invoke AUTOSAR Server Operation Block in Referenced Model” on page 22-63
“Server Operation and Model Block with States” on page 22-63
“Cannot Save Importer Objects in MAT-Files” on page 22-63
“Using the Merge Block for Inter-Runnable Variables” on page 22-63
“AUTOSAR Compiler Abstraction Macros” on page 22-64
“Intrinsic Fixed-Point Types for Model Configured as Server” on page 22-64
“Server Operation Model with Tunable Parameters” on page 22-65
“Migrating AUTOSAR Development Kit Models” on page 22-65

Cannot Import Internal Behavior

Internal behavior is not parsed. This means any I/O information stored at the runnable level (for example, implicit or explicit) is not imported, and all internal I/O settings default to implicit. You can subsequently configure these I/O ports with the `setIODataAccessMode` method or in the Configure AUTOSAR Interface dialog box.

Cannot Copy Subsystem Blocks Without Losing Interface Information

If you copy and paste a subsystem block to create a new block in either a new model or the same model, the AUTOSAR interface information stored with the original subsystem block does not copy to the new subsystem block.

Error If No Default Configuration

To avoid build errors, do not clear the **Generate code only** check box. Configure your model using the **Get Default Configuration** button or the `RTW.AutosarInterface.getDefaultConf` method. If you try to build an executable with the AUTOSAR target without supplying your own system target file or generating a software-in-the-loop (SIL) S-function, you see an error.

Server Operation Model with Reusable Function Subsystems

The software does not provide AUTOSAR support for a model that is configured as a server operation and contains subsystems that have the reusable function code generation option selected (see “Reusable Function Option” and “Generating Reusable Code for Subsystems Containing S-Function Blocks”).

Invoke AUTOSAR Server Operation Block in Referenced Model

The software does not support the use of the Invoke AUTOSAR Server Operation block in a referenced model.

Server Operation and Model Block with States

The software does not provide AUTOSAR support for a model that is configured as a server operation and has model blocks.

Cannot Save Importer Objects in MAT-Files

If you try to save an `arxml.importer` object in a MAT-file, you lose all the information. If you reload the MAT-file, then the object is null (`handle = -1`), because of the Java™ objects that compose the `arxml.importer` object.

Using the Merge Block for Inter-Runnable Variables

You can use the Merge block to merge inter-runnable variables. However, you must do the following:

- Ensure that the output signal of the Merge block is connected to either one root output or one or more subsystems.
- If the output signal of the Merge block is connected to the inputs of one or more subsystems, assign the same signal name to the Merge block's output and inputs.

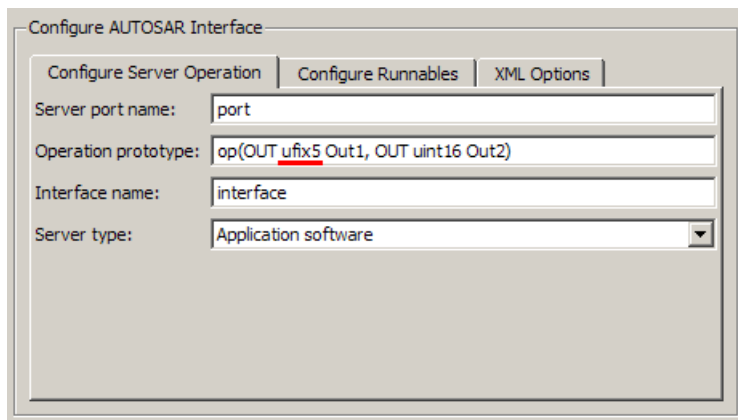
AUTOSAR Compiler Abstraction Macros

The software does not generate AUTOSAR compiler abstraction macros for data or functions arising from the following:

- Model blocks
- Stateflow
- Embedded MATLAB
- Shared utility functions
- Custom storage classes
- Local or temporary variables

Intrinsic Fixed-Point Types for Model Configured as Server

The software does not support operation prototype arguments with intrinsic fixed-point data types. For example, `ufix5` shown in the following figure.



The software produces an error when you build the model.

To work around this limitation, before building the model, create a `Simulink.NumericType` base workspace object with the required property values. For example, to create a `Simulink.NumericType` object `ufix5`, enter the following in the Command Window:

```
>> ufix5 = Simulink.NumericType;
>> ufix5.DataTypeMode = 'Fixed-point: binary point scaling';
>> ufix5.Signedness = 'Signed'
>> ufix5.WordLength = 16
>> ufix5.FractionLength = 0
>> ufix5.IsAlias = 1;
>> ufix.HeaderFile = 'Rte_Type.h'
```

For more information, see `Simulink.NumericType` in the Simulink documentation.

Server Operation Model with Tunable Parameters

The software does not provide AUTOSAR support for a model that is configured as a server operation and has tunable parameters with storage class set to `SimulinkGlobal (Auto)`.

Migrating AUTOSAR Development Kit Models

Use the `autosar_adk_migrate` function to migrate an AUTOSAR Development Kit (ADK) model (from releases before R2008a) to the AUTOSAR interface.

Enter:

```
autosar_adk_migrate(PATHNAME)
```

to migrate the ADK model/system specified by the full path name `PATHNAME` from the ADK settings to the new AUTOSAR interface. The model must be open before you invoke this function. The MathWorks recommends that you save the migrated model to a different file name.

Demos and Further Reading

AUTOSAR Demos

For detailed explanations of AUTOSAR workflows with Real-Time Workshop Embedded Coder software, see the demos in the following table.

Demo	Description
AUTOSAR Code Generation: rtwdemo_autosar_legacy_script	How to generate AUTOSAR-compliant code and export AUTOSAR Software Component description XML files from a Simulink model
Using an AUTOSAR Client-Server Interface rtwdemo_autosar_clientserver_script	How to configure and generate AUTOSAR-compliant code and export AUTOSAR-compliant XML files for a Simulink model with an AUTOSAR client-server interface
AUTOSAR Code Generation for Multiple Runnable Entities: rtwdemo_autosar_multirunnables_script	How to configure and generate AUTOSAR-compliant code and export AUTOSAR Software Component description XML files for a Simulink model with multiple runnables.
Import and Export an AUTOSAR Software Component: rtwdemo_autosar_roundtrip_script	How to use an AUTOSAR authoring tool with Simulink to develop AUTOSAR Software Components. Learn how to import software component interfaces into Simulink, modify and export them, and merge the completed software component back into an AUTOSAR authoring tool.

Further Reading

For more information, see the AUTOSAR Web site:
<http://www.autosar.org/>

Integrating External Code and Generated C and C++ Code

- Chapter 23, “About External Code Integration Extensions”
- Chapter 24, “Generating S-Function Wrappers”
- Chapter 25, “Exporting Function-Call Subsystems”
- Chapter 26, “Nonvirtual Subsystem Modular Function Code Generation”
- Chapter 27, “Controlling Generation of Function Prototypes”
- Chapter 28, “Controlling Generation of Encapsulated C++ Model Interfaces”
- Chapter 29, “Replacing Math Functions and Operators Using Target Function Libraries”

About External Code Integration Extensions

The Real-Time Workshop documentation introduces capabilities for integrating external code with generated C and C++ code. Topics include

- “About External Code Integration”
- “Integrating External Code Using Model Configuration Parameters”
- “Integrating External Code Using Custom Code Blocks”
- “Integrating External Code Using S-Functions”

The Real-Time Workshop Embedded Coder product extends the preceding capabilities to support

- Chapter 24, “Generating S-Function Wrappers”
- Chapter 25, “Exporting Function-Call Subsystems”
- Chapter 26, “Nonvirtual Subsystem Modular Function Code Generation”
- Chapter 27, “Controlling Generation of Function Prototypes”
- Chapter 28, “Controlling Generation of Encapsulated C++ Model Interfaces”
- Chapter 29, “Replacing Math Functions and Operators Using Target Function Libraries”

Generating S-Function Wrappers

- “About S-Function Wrapper Generation” on page 24-2
- “Creating a SIL Block” on page 24-4
- “S-Function Wrapper Generation Limitations” on page 24-6

About S-Function Wrapper Generation

An S-function wrapper is an S-function that calls your C or C++ code from within Simulink. S-function wrappers provide a standard interface between Simulink and externally written code, allowing you to integrate your code into a model with minimal modification. This is useful for software-in-the-loop (SIL) code verification (validating your generated code in Simulink), as well as for simulation acceleration purposes (see “Verifying Code Using a SIL Block” on page 36-30). For a complete description of wrapper S-functions, see the Simulink Writing S-Functions document.

Using the Real-Time Workshop Embedded Coder **Create SIL block** option, you can build, in one automated step:

- A noninlined C or C++ MEX S-function wrapper that calls Real-Time Workshop Embedded Coder generated code
- A model with a SIL block. This block, which contains the generated S-function, is ready for use with other blocks or models

When the **Create SIL block** option is selected, the Real-Time Workshop build process generates an additional source code file, *model_sf.c* or *.cpp*, in the build directory. This module contains the S-function that calls the Real-Time Workshop Embedded Coder code that you deploy. You can use this S-function within Simulink.

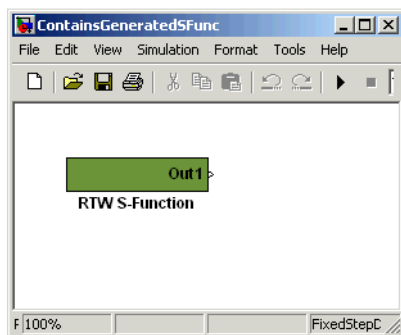
The build process then compiles and links *model_sf.c* or *.cpp* with *model.c* or *.cpp* and the other Real-Time Workshop Embedded Coder generated code modules, building a MEX-file. The MEX-file is named *model_sf.mexext*. (*mexext* is the file extension for MEX-files on your platform, as given by the MATLAB *mexext* command.) The MEX-file is stored in your working directory. Finally, the Real-Time Workshop build process creates and opens an untitled model containing the SIL block with the generated S-function.

Note To generate a wrapper S-function for a subsystem, you can use a right-click subsystem build. Right-click the subsystem block in your model, select **Real-Time Workshop > Generate S-Function**, and in the Generate S-Function dialog box, select **Create Real-Time Workshop Embedded Coder SIL block** and click **Build**.

Creating a SIL Block

To create a SIL block with the S-function wrapper for your Real-Time Workshop Embedded Coder code, open your ERT-based Simulink model and do the following:

- 1 Open the Configuration Parameters dialog box.
- 2 Select the **SIL and PIL Verification** pane.
- 3 Select the **Create SIL block** check box.
- 4 Configure the other code generation options as required.
- 5 To ensure that memory for the S-function is initialized to zero, you must deselect the following options in the **Optimization > Data Initialization** pane:
 - “**Remove root level I/O zero initialization**”
 - “**Remove internal data zero initialization**”
 - “**Use memset to initialize floats and doubles to 0.0**”
- 6 Select the **Real-Time Workshop** pane, and click **Build**.
- 7 When the build process completes, an untitled model window opens. This model contains a SIL block with the generated S-function.



- 8 Save the new model.

9 You can now use the SIL block with other blocks or models in Simulink.

S-Function Wrapper Generation Limitations

The following limitations apply to Real-Time Workshop Embedded Coder S-function wrapper generation:

- Continuous sample time is not supported. The **Support continuous time** option should not be selected when creating a Real-Time Workshop Embedded Coder SIL block.
- Models that contain S-function blocks for which the S-function is not inlined with a TLC file are not supported when creating a Real-Time Workshop Embedded Coder SIL block.
- You cannot use multiple instances of a Real-Time Workshop Embedded Coder SIL block within a model, because the code uses static memory allocation. Each instance potentially can overwrite global data values of the others.
- Real-Time Workshop Embedded Coder SIL blocks can be used with other blocks and models for SIL code verification and simulation acceleration, but they cannot be used for code generation.
- A MEX S-function wrapper must only be used in the version of MATLAB in which the wrapper is created.

Exporting Function-Call Subsystems

- “Overview” on page 25-2
- “Requirements for Exporting Function-Call Subsystems” on page 25-4
- “Techniques for Exporting Function-Call Subsystems” on page 25-7
- “Optimizing Exported Function-Call Subsystems” on page 25-8
- “Exporting Function-Call Subsystems That Depend on Elapsed Time” on page 25-9
- “Function-Call Subsystem Export Example” on page 25-10
- “Function-Call Subsystems Export Limitations” on page 25-14

Overview

Real-Time Workshop Embedded Coder software provides code export capabilities that you can use to

- Automatically generate code for
 - A function-call subsystem that contains only blocks that support code generation
 - A virtual subsystem that contains only such subsystems and a few other types of blocks
- Create a SIL block that represents the generated code

You can use these capabilities only if the subsystem and its interface to the Simulink model conform to certain requirements and constraints, as described in “Requirements for Exporting Function-Call Subsystems” on page 25-4. For limitations that apply, see “Function-Call Subsystems Export Limitations” on page 25-14.

Note For models designed in earlier releases, Real-Time Workshop Embedded Coder software also supports the ability to export functions from triggered subsystems. In general, the requirements and limitations stated for exporting functions from function-call subsystems also apply to exporting functions from triggered subsystems, with the following exceptions:

- Triggered subsystems from which you intend to export functions must be encapsulated in a single top-level virtual subsystem.
 - Triggered subsystems do not have to meet the requirements in “Trigger Signals Require a Common Source” on page 25-5 and “Requirements for Exported Virtual Subsystems” on page 25-5.
 - The section “Exporting Function-Call Subsystems That Depend on Elapsed Time” on page 25-9 is not applicable to exporting functions from triggered subsystems.
-

Exported Subsystems Demo

To see a demo of exported function-call subsystems, type `rtwdemo_export_functions` in the MATLAB Command Window.

Additional Information

See the following in the Simulink documentation for additional information relating to exporting function-call subsystems:

- “Systems and Subsystems”
- “Signals”
- “Triggered Subsystems”
- “Function-Call Subsystems”
- *Developing S-Functions*

If you want to use Stateflow blocks to trigger exportable function-call subsystems, you may also need information from the *Stateflow and Stateflow® Coder™ User’s Guide*.

Requirements for Exporting Function-Call Subsystems

To be exportable as code, a function-call subsystem, or a virtual subsystem that contains such subsystems, must meet certain requirements. Most requirements are similar for either type of export, but some apply only to virtual subsystems. The requirements that affect all Simulink code generation also apply.

For brevity, *exported subsystem* in this section means only an exported function-call subsystem or an exported virtual subsystem that contains such subsystems. The requirements listed do not necessarily apply to other types of exported subsystems.

Requirements for All Exported Subsystems

These requirements apply to both exported function-call subsystems and exported virtual subsystems that contain such subsystems.

Blocks Must Support Code Generation

All blocks within an exported subsystem must support code generation. However, blocks outside the subsystem need not support code generation unless they will be converted to code in some other context.

Blocks Must Not Use Absolute Time

Certain blocks use absolute time. Blocks that use absolute time are not supported in exported function-call subsystems. For a complete list of such blocks, see “Limitations on the Use of Absolute Time” in the Real-Time Workshop documentation.

Blocks Must Not Depend on Elapsed Time

Certain blocks, like the Sine Wave block and Discrete Integrator block, depend on elapsed time. If an exported function-call subsystem contains any blocks that depend on elapsed time, the subsystem must specify periodic execution. See “Exporting Function-Call Subsystems That Depend on Elapsed Time” on page 25-9 in the Real-Time Workshop documentation.

Trigger Signals Require a Common Source

If more than one trigger signal crosses the boundary of an exported system, all of the trigger signals must be periodic and originate from the same function-call initiator.

Trigger Signals Must Be Scalar

A trigger signal that crosses the boundary of an exported subsystem must be scalar. Input and output data signals that do not act as triggers need not be scalar.

Data Signals Must Be Nonvirtual

A data signal that crosses the boundary of an exported system cannot be a virtual bus, and cannot be implemented as a Goto-From connection. Every data signal crossing the export boundary must be scalar, muxed, or a nonvirtual bus.

Requirements for Exported Virtual Subsystems

These requirements apply only to exported virtual subsystems that contain function-call subsystems.

Virtual Subsystem Must Use Only Permissible Blocks

The top level of an exported virtual subsystem that contains function-call subsystem blocks can contain only the following other types of blocks:

- Input and Output blocks (ports)
- Constant blocks (including blocks that resolve to constants, such as Add)
- Merge blocks
- Virtual connection blocks (Mux, Demux, Bus Creator, Bus Selector, Signal Specification)
- Signal-viewer blocks, such as Scope blocks

These restrictions do *not* apply within function-call subsystems, whether or not they appear in a virtual subsystem. They apply only at the top level

of an exported virtual subsystem that contains one or more function-call subsystems.

Constant Blocks Must Be Inlined

When a constant block appears at the top level of an exported virtual subsystem, the containing model must check `Inline` parameters on the **Optimization** pane of the Configuration Parameters dialog box.

Constant Outputs Must Specify a Storage Class

When a constant signal drives an output port of an exported virtual subsystem, the signal must specify a storage class.

Techniques for Exporting Function-Call Subsystems

To export a function-call subsystem, or a virtual subsystem that contains function-call subsystems,

- 1** Ensure that the subsystem to be exported satisfies the “Requirements for Exporting Function-Call Subsystems” on page 25-4.
- 2** In the Configuration Parameters dialog box:
 - a** On the **Real-Time Workshop** pane, specify an ERT code generation target such as `ert.tlc`.
 - b** If you want a SIL block with the generated code, go to the **SIL and PIL Verification** pane and select **Create SIL block**.
 - c** Click **OK** or **Apply**.
- 3** Right-click the subsystem block and choose **Real-Time Workshop > Export Functions** from the context menu.

The **Build code for subsystem: *Subsystem*** dialog box appears. This dialog box is not specific to exporting function-call subsystems, and generating code does not require entering information in the box.

- 4** Click **Build**.

The MATLAB Command Window displays messages similar to those for any code generation sequence. Simulink generates code and places it in the working directory.

If you checked **Create SIL block** in step 2b, Simulink opens a new window that contains an S-function block that represents the generated code. This block has the same size, shape, and connectors as the original subsystem.

Code generation and optional block creation are now complete. You can test and use the code and optional block as you could any generated ERT code and S-function block.

Optimizing Exported Function-Call Subsystems

You can use Real-Time Workshop options to optimize the code generated for a function-call subsystem or virtual block that contains such subsystems. To obtain faster code,

- Specify a storage class for every input signal and output signal that crosses the boundary of the subsystem.
- For each function-call subsystem to be exported (whether directly or within a virtual subsystem):
 - 1** Right-click the subsystem and choose **Subsystem Parameters** from the context menu.
 - 2** Set the **Real-Time Workshop system code** parameter to Auto.
 - 3** Click **OK** or **Apply**.

Exporting Function-Call Subsystems That Depend on Elapsed Time

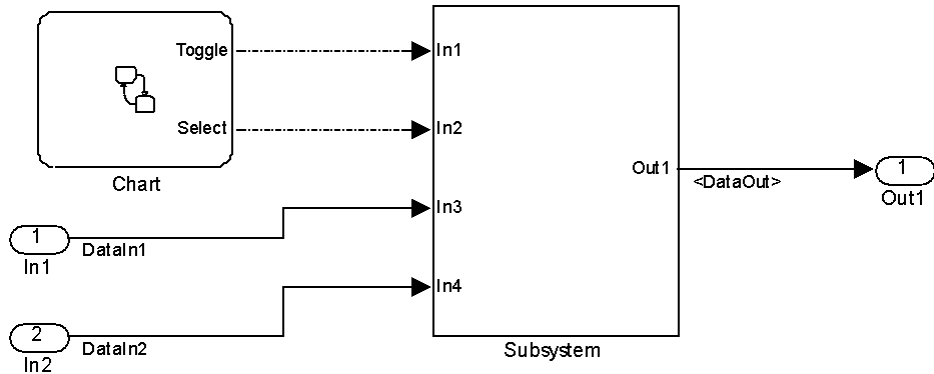
Some blocks, such as the Sine Wave block (if sample-based) and the Discrete-Time Integrator block, depend on elapsed time. See “Absolute and Elapsed Time Computation” in the Real-Time Workshop documentation for more information.

When a block that depends on elapsed time exists in a function-call subsystem, the subsystem cannot be exported unless it specifies periodic execution. To provide the necessary specification,

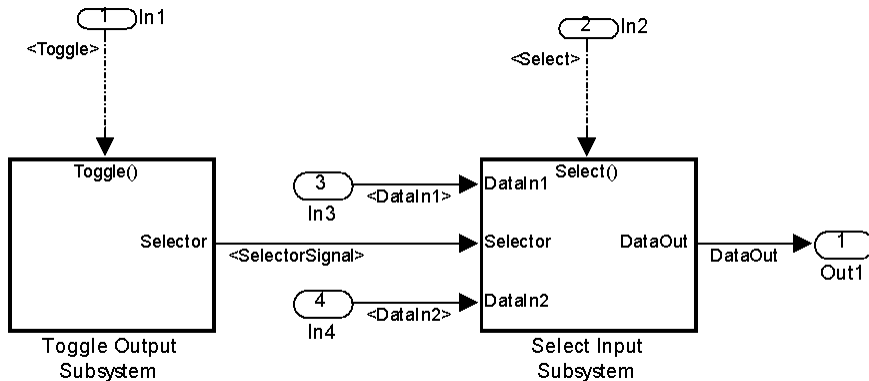
- 1** Right-click the trigger port block in the function-call subsystem and choose **TriggerPort Parameters** from the context menu.
- 2** Specify **periodic** in the **Sample time type** field.
- 3** Set the **Sample time** to the same granularity specified (directly or by inheritance) in the function-call initiator.
- 4** Click **OK** or **Apply**.

Function-Call Subsystem Export Example

The next figure shows the top level of a model that uses a Stateflow chart named Chart to input two function-call trigger signals (denoted by dash-dot lines) to a virtual subsystem named Subsystem.



The next figure shows the contents of Subsystem in the previous figure. The subsystem contains two function-call subsystems, each driven by one of the signals input from the top level.



In the preceding model, the Stateflow chart can assert either of two scalar signals, Toggle and Select.

- Asserting `Toggle` toggles the Boolean state of the function-call subsystem `Toggle Output Subsystem`.
- Asserting `Select` causes the function-call subsystem `Select Input Subsystem` to assign the value of `DataIn1` or `DataIn2` to its output signal. The value assigned depends on the current state of `Toggle Output Subsystem`.

The following generated code implements the subsystem named `Subsystem`. The code is typical for virtual subsystems that contain function-call subsystems. It specifies an initialization function and a function for each contained subsystem, and would also include functions to enable and disable subsystems if applicable.

```
#include "Subsystem.h"
#include "Subsystem_private.h"

/* Exported block signals */
real_T DataIn1;           /* '<Root>/In3' */
real_T DataIn2;          /* '<Root>/In4' */
real_T DataOut;          /* '<S4>/Switch' */
boolean_T SelectorSignal; /* '<S5>/Logical Operator' */

/* Exported block states */
boolean_T SelectorState; /* '<S5>/Unit Delay' */

/* Real-time model */
RT_MODEL_Subsystem Subsystem_M_;
RT_MODEL_Subsystem *Subsystem_M = &Subsystem_M_;

/* Initial conditions for exported function: Toggle */

void Toggle_Init(void)
{
    /* Initial conditions for function-call system: '<S1>/Toggle Output Subsystem' */

    /* InitializeConditions for UnitDelay: '<S5>/Unit Delay' */
    SelectorState = Subsystem_P.UnitDelay_X0;
}

/* Output and update for exported function: Toggle */
```

```
void Toggle(void)
{
    /* Output and update for function-call system: '<S1>/Toggle Output Subsystem' */

    /* Logic: '<S5>/Logical Operator' incorporates:
     * UnitDelay: '<S5>/Unit Delay'
     */
    SelectorSignal = !SelectorState;

    /* Update for UnitDelay: '<S5>/Unit Delay' */
    SelectorState = SelectorSignal;
}

/* Output and update for exported function: Select */

void Select(void)
{
    /* Output and update for function-call system: '<S1>/Select Input Subsystem' */

    /* Switch: '<S4>/Switch' incorporates:
     * Inport: '<Root>/In3'
     * Inport: '<Root>/In4'
     */
    if(SelectorSignal) {
        DataOut = DataIn1;
    } else {
        DataOut = DataIn2;
    }
}

/* Model initialize function */

void Subsystem_initialize(void)
{
    /* initialize error status */
    rtmSetErrorStatus(Subsystem_M, (const char_T *)0);

    /* block I/O */
}
```

```
/* exported global signals */
DataOut = 0.0;
SelectorSignal = FALSE;

/* states (dwork) */

/* exported global states */
SelectorState = FALSE;

/* external inputs */
DataIn1 = 0.0;
DataIn2 = 0.0;

Toggle_Init();
}

/* Model terminate function */

void Subsystem_terminate(void)
{
    /* (no terminate code required) */
}
```

Function-Call Subsystems Export Limitations

The function-call subsystem export capabilities have the following limitations:

- Real-Time Workshop options do not control the names of the files containing the generated code. All such filenames begin with the name of the exported subsystem. Each filename is suffixed as appropriate to the file.
- Real-Time Workshop options do not control the names of top-level functions in the generated code. Each function name reflects the name of the signal that triggers the function, or for an unnamed signal, the block from which the signal originates.
- The software cannot export reusable code for a function-call subsystem. Checking **Configuration Parameters > Real-Time Workshop > Interface > Generate reusable code** has no effect on the generated code for the subsystem.
- The software supports code generation for a SIL block provided the block does not have function-call input ports. However, the block will appear as a noninlined S-function in the generated code.
- The software supports a SIL block in accelerator mode only if its function-call initiator is noninlined in accelerator mode. Examples of noninlined initiators include all Stateflow charts.
- The SIL block must be driven by a Level-2 S-function initiator block, such as a Stateflow chart or the built-in Function-call Generator block.
- An asynchronous (sample-time) function-call system can be exported, but the software does not support the ERT S-function wrapper for an asynchronous system.
- The software does not support code generation for a SIL block if the block is generated for exported function calls.
- The output from a SIL block cannot be merged using the Merge block.
- The software does not support MAT-file logging for exported function calls. Any specification that enables MAT-file logging is ignored.
- The use of the TLC function `LibIsFirstInit` is deprecated for exported function calls.

- The `model_initialize` function generated in the code for an exported function-call subsystem never includes a `firstTime` argument, regardless of the value of the model configuration parameter `IncludeERTFirstTime`. Thus, you cannot call `model_initialize` at a time greater than start time, for example, to reset block states.

Nonvirtual Subsystem Modular Function Code Generation

- “Overview” on page 26-2
- “Configuring Nonvirtual Subsystems for Generating Modular Function Code” on page 26-4
- “Examples of Modular Function Code for Nonvirtual Subsystems” on page 26-9
- “Nonvirtual Subsystem Modular Function Code Limitations” on page 26-15

Overview

The Real-Time Workshop Embedded Coder software provides a subsystem option, **Function with separate data**, that allows you to generate modular function code for nonvirtual subsystems, including atomic subsystems and conditionally executed subsystems.

By default, the generated code for a nonvirtual subsystem does not separate a subsystem's internal data from the data of its parent Simulink model. This can make it difficult to trace and test the code, particularly for nonreusable subsystems. Also, in large models containing nonvirtual subsystems, data structures can become large and potentially difficult to compile.

The Subsystem Parameters dialog box option **Function with separate data** allows you to generate subsystem function code in which the internal data for a nonvirtual subsystem is separated from its parent model and is owned by the subsystem. As a result, the generated code for the subsystem is easier to trace and test. The data separation also tends to reduce the size of data structures throughout the model.

Note Selecting the **Function with separate data** option for a nonvirtual subsystem has no semantic effect on the parent Simulink model.

To be able to use this option,

- Your Simulink model must use an ERT-based system target file (requires a Real-Time Workshop Embedded Coder license).
- Your subsystem must be configured to be atomic or conditionally executed (for more information, see “Systems and Subsystems” in the Simulink documentation).
- Your subsystem must use the **Function** setting for the **Real-Time Workshop system code** parameter.

To configure your subsystem for generating modular function code, you invoke the Subsystem Parameters dialog box and make a series of selections to display and enable the **Function with separate data** option. See “Configuring Nonvirtual Subsystems for Generating Modular Function

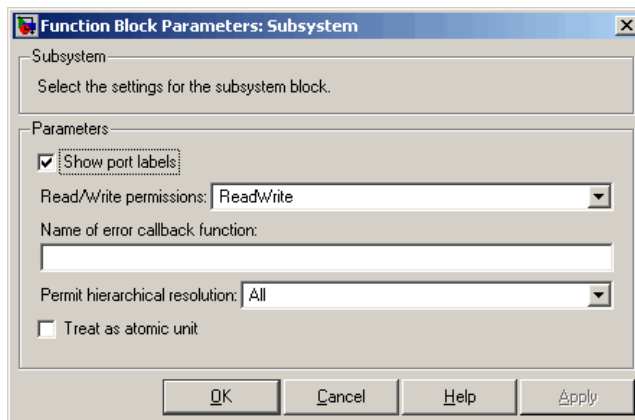
Code” on page 26-4 and “Examples of Modular Function Code for Nonvirtual Subsystems” on page 26-9 for details. For limitations that apply, see “Nonvirtual Subsystem Modular Function Code Limitations” on page 26-15.

For more information about generating code for atomic subsystems, see the sections “Creating Subsystems” and “Generating Code and Executables from Subsystems” in the Real-Time Workshop documentation.

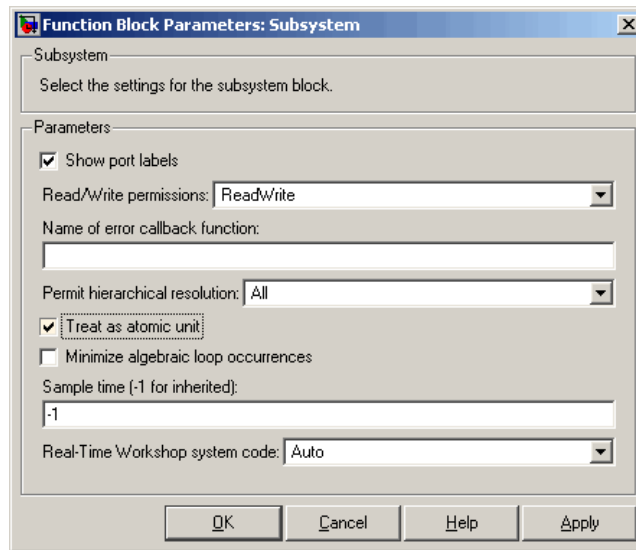
Configuring Nonvirtual Subsystems for Generating Modular Function Code

This section summarizes the steps needed to configure a subsystem in a Simulink model for modular function code generation.

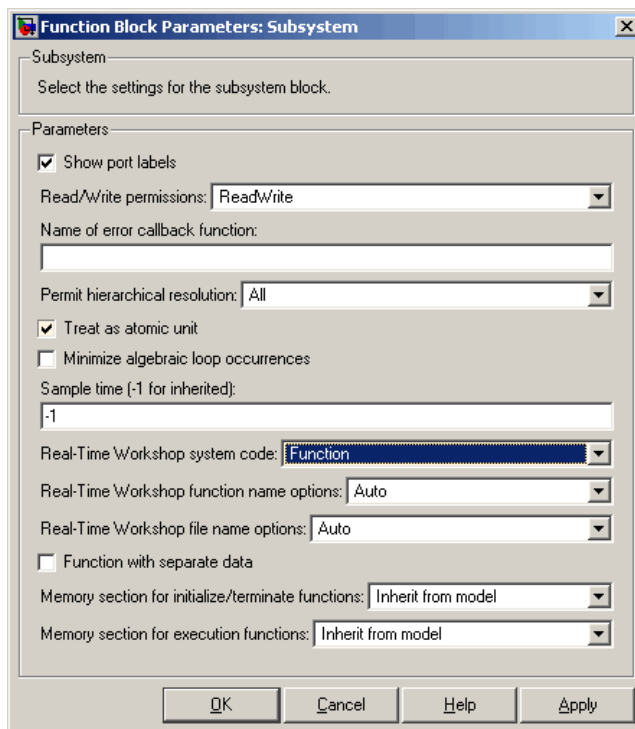
- 1 Verify that the Simulink model containing the subsystem uses an ERT-based system target file (see the **System target file** parameter on the **Real-Time Workshop** pane of the Configuration Parameters dialog box).
- 2 In your Simulink model, select the subsystem for which you want to generate modular function code and launch the Subsystem Parameters dialog box (for example, right-click the subsystem and select **Subsystem Parameters**). The dialog box for an atomic subsystem is shown below. (In the dialog box for a conditionally executed subsystem, the dialog box option **Treat as atomic unit** is greyed out, and you can skip Step 3.)



- 3 If the Subsystem Parameters dialog box option **Treat as atomic unit** is available for selection but not selected, the subsystem is neither atomic nor conditionally executed. Select the option **Treat as atomic unit**. After you make this selection, the **Real-Time Workshop system code** parameter is displayed.

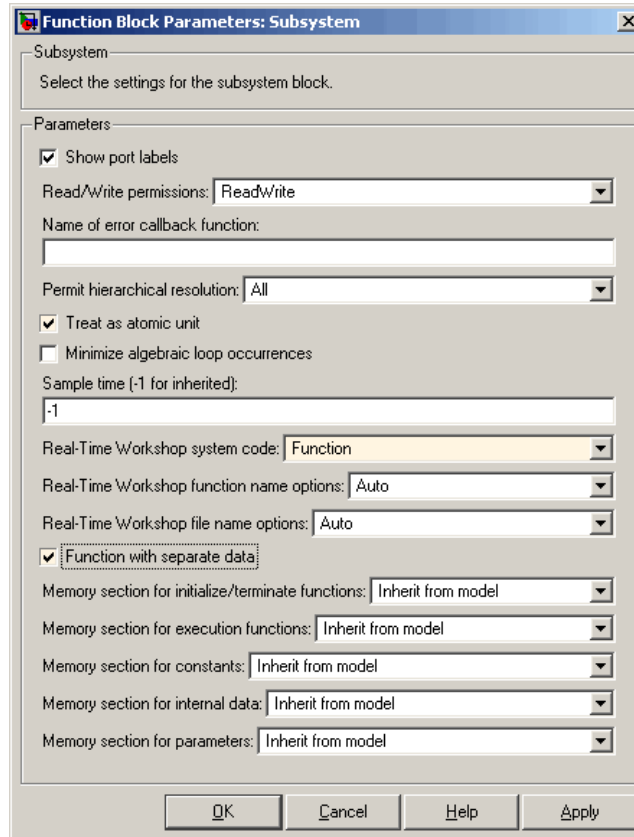


- 4 For the **Real-Time Workshop system code** parameter, select the value **Function**. After you make this selection, the **Function with separate data** option is displayed.



Note Before you generate nonvirtual subsystem function code with the **Function with separate data** option selected, you might want to generate function code with the option *deselected* and save the generated function .c and .h files in a separate directory for later comparison.

- 5 Select the **Function with separate data** option. After you make this selection, additional configuration parameters are displayed.



Note To control the naming of the subsystem function and the subsystem files in the generated code, you can modify the subsystem parameters **Real-Time Workshop function name options** and **Real-Time Workshop file name options**.

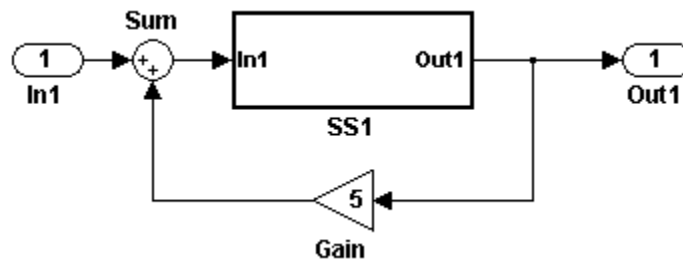
- 6 To save your subsystem parameter settings and exit the dialog box, click **OK**.

This completes the subsystem configuration needed to generate modular function code. You can now generate the code for the subsystem and examine the generated files, including the function `.c` and `.h` files named according to your subsystem parameter specifications. For more information on generating code for nonvirtual subsystems, see “Creating Subsystems” in the Real-Time Workshop documentation. For examples of generated subsystem function code, see “Examples of Modular Function Code for Nonvirtual Subsystems” on page 26-9.

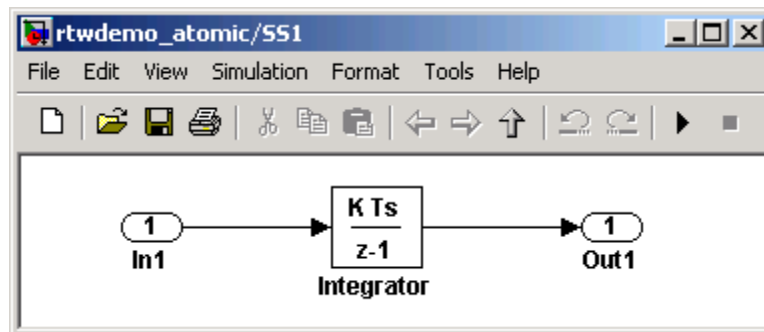
Examples of Modular Function Code for Nonvirtual Subsystems

To illustrate the effect of selecting the **Function with separate data** option for a nonvirtual subsystem, the following procedure generates atomic subsystem function code with and without the option selected and compares the results.

- 1 Open MATLAB and launch `rtwdemo_atomic.mdl` using the MATLAB command `rtwdemo_atomic`. Examine the Simulink model.



- 2 Double-click the SS1 subsystem and examine the contents. (You can close the subsystem window when you are finished.)



- 3 Use the Configuration Parameters dialog box to change the model's **System target file** from GRT to ERT. For example, from the Simulink window, select **Simulation > Configuration Parameters**, select the **Real-Time**

Workshop pane, select **System target file** `ert.tlc`, and click **OK** twice to confirm the change.

- 4** Create a variant of `rtwdemo_atomic.mdl` that illustrates function code *without* data separation.
 - a** In the Simulink view of `rtwdemo_atomic.mdl`, right-click the SS1 subsystem and select **Subsystem Parameters**. In the Subsystem Parameters dialog box, verify that
 - **Treat as atomic unit** is checked
 - **User specified** is selected as the value for the **Real-Time Workshop function name options** parameter
 - `myfun` is specified as the value for the **Real-Time Workshop function name** parameter
 - b** In the Subsystem Parameters dialog box,
 - i** Select the value **Function** for the **Real-Time Workshop system code** parameter. After this selection, additional parameters and options will appear.
 - ii** Select the value **Use function name** for the **Real-Time Workshop file name options** parameter. This selection is optional but simplifies the later task of code comparison by causing the atomic subsystem function code to be generated into the files `myfun.c` and `myfun.h`.

Do *not* select the option **Function with separate data**. Click **Apply** to apply the changes and click **OK** to exit the dialog box.

 - c** Save this model variant to a personal work directory, for example, `d:/atomic/rtwdemo_atomic1.mdl`.
- 5** Create a variant of `rtwdemo_atomic.mdl` that illustrates function code *with* data separation.
 - a** In the Simulink view of `rtwdemo_atomic1.mdl` (or `rtwdemo_atomic.mdl` with step 3 reapplied), right-click the SS1 subsystem and select **Subsystem Parameters**. In the Subsystem Parameters dialog box, verify that
 - **Treat as atomic unit** is checked

- Function is selected for the **Real-Time Workshop system code** parameter
 - User specified is selected as the value for the **Real-Time Workshop function name options** parameter
 - myfun is specified as the value for the **Real-Time Workshop function name** parameter
 - Use function name is selected for the **Real-Time Workshop file name options** parameter
- b** In the Subsystem Parameters dialog box, select the option **Function with separate data**. Click **Apply** to apply the change and click **OK** to exit the dialog box.
 - c** Save this model variant, using a different name than the first variant, to a personal work directory, for example, `d:/atomic/rtwdemo_atomic2.mdl`.
- 6** Generate code for each model, `d:/atomic/rtwdemo_atomic1.mdl` and `d:/atomic/rtwdemo_atomic2.mdl`.
 - 7** In the generated code directories, compare the `model.c/.h` and `myfun.c/.h` files generated for the two models. (In this example, there are no significant differences in the generated variants of `ert_main.c`, `model_private.h`, `model_types.h`, or `rtwtypes.h`.)

H File Differences for Nonvirtual Subsystem Function Data Separation

The differences between the H files generated for `rtwdemo_atomic1.mdl` and `rtwdemo_atomic2.mdl` help illustrate the effect of selecting the **Function with separate data** option for nonvirtual subsystems.

- 1** Selecting **Function with separate data** causes typedefs for subsystem data to be generated in the `myfun.h` file for `rtwdemo_atomic2`:

```

/* Block signals for system '<Root>/SS1' */
typedef struct {
    real_T Integrator;           /* '<S1>/Integrator' */
} rtB_myfun;

/* Block states (auto storage) for system '<Root>/SS1' */

```

```
typedef struct {
    real_T Integrator_DSTATE;          /* '<S1>/Integrator' */
} rtDW_myfun;
```

By contrast, for `rtwdemo_atomic1`, typedefs for subsystem data belong to the model and appear in `rtwdemo_atomic1.h`:

```
/* Block signals (auto storage) */
typedef struct {
    ...
    real_T Integrator;                /* '<S1>/Integrator' */
} BlockIO_rtwdemo_atomic1;

/* Block states (auto storage) for system '<Root>' */
typedef struct {
    real_T Integrator_DSTATE;        /* '<S1>/Integrator' */
} D_Work_rtwdemo_atomic1;
```

2 Selecting **Function with separate data** generates the following external declarations in the `myfun.h` file for `rtwdemo_atomic2`:

```
/* Extern declarations of internal data for 'system '<Root>/SS1'' */
extern rtB_myfun rtwdemo_atomic2_myfunB;

extern rtDW_myfun rtwdemo_atomic2_myfunDW;

extern void myfun_initialize(void);
```

By contrast, the generated code for `rtwdemo_atomic1` contains model-level external declarations for the subsystem's `BlockIO` and `D_Work` data, in `rtwdemo_atomic1.h`:

```
/* Block signals (auto storage) */
extern BlockIO_rtwdemo_atomic1 rtwdemo_atomic1_B;

/* Block states (auto storage) */
extern D_Work_rtwdemo_atomic1 rtwdemo_atomic1_DWork;
```

C File Differences for Nonvirtual Subsystem Function Data Separation

The differences between the C files generated for `rtwdemo_atomic1.mdl` and `rtwdemo_atomic2.mdl` illustrate the key effects of selecting the **Function with separate data** option for nonvirtual subsystems.

- 1 Selecting **Function with separate data** causes a separate subsystem initialize function, `myfun_initialize`, to be generated in the `myfun.c` file for `rtwdemo_atomic2`:

```
void myfun_initialize(void) {
    {
        ((real_T*)&rtwdemo_atomic2_myfunB.Integrator)[0] = 0.0;
    }
    rtwdemo_atomic2_myfunDW.Integrator_DSTATE = 0.0;
}
```

The subsystem initialize function in `myfun.c` is invoked by the model initialize function in `rtwdemo_atomic2.c`:

```
/* Model initialize function */

void rtwdemo_atomic2_initialize(void)
{
    ...

    /* Initialize subsystem data */
    myfun_initialize();
}
```

By contrast, for `rtwdemo_atomic1`, subsystem data is initialized by the model initialize function in `rtwdemo_atomic1.c`:

```
/* Model initialize function */

void rtwdemo_atomic1_initialize(void)
{
    ...
    /* block I/O */
    {
```

```

...
    ((real_T*)&rtwdemo_atomic1_B.Integrator)[0] = 0.0;
}

/* states (dwork) */

rtwdemo_atomic1_DWork.Integrator_DSTATE = 0.0;
...
}

```

- 2** Selecting **Function with separate data** generates the following declarations in the `myfun.c` file for `rtwdemo_atomic2`:

```

/* Declare variables for internal data of system '<Root>/SS1' */
rtB_myfun rtwdemo_atomic2_myfunB;

rtDW_myfun rtwdemo_atomic2_myfunDW;

```

By contrast, the generated code for `rtwdemo_atomic1` contains model-level declarations for the subsystem's `BlockIO` and `D_Work` data, in `rtwdemo_atomic1.c`:

```

/* Block signals (auto storage) */
BlockIO_rtwdemo_atomic1 rtwdemo_atomic1_B;

/* Block states (auto storage) */
D_Work_rtwdemo_atomic1 rtwdemo_atomic1_DWork;

```

- 3** Selecting **Function with separate data** generates identifier naming that reflects the subsystem orientation of data items. Notice the references to subsystem data in subsystem functions such as `myfun` and `myfun_update` or in the model's `model_step` function. For example, compare this code from `myfun` for `rtwdemo_atomic2`

```

/* DiscreteIntegrator: '<S1>/Integrator' */
rtwdemo_atomic2_myfunB.Integrator = rtwdemo_atomic2_myfunDW.Integrator_DSTATE;

```

to the corresponding code from `myfun` for `rtwdemo_atomic1`.

```

/* DiscreteIntegrator: '<S1>/Integrator' */
rtwdemo_atomic1_B.Integrator = rtwdemo_atomic1_DWork.Integrator_DSTATE;

```

Nonvirtual Subsystem Modular Function Code Limitations

The nonvirtual subsystem option **Function with separate data** has the following limitations:

- The **Function with separate data** option is available only in ERT-based Simulink models (requires a Real-Time Workshop Embedded Coder license).
- The nonvirtual subsystem to which the option is applied cannot have multiple sample times or continuous sample times; that is, the subsystem must be single-rate with a discrete sample time.
- The nonvirtual subsystem cannot contain continuous states.
- The nonvirtual subsystem cannot output function call signals.
- The nonvirtual subsystem cannot contain noninlined S-functions.
- The generated files for the nonvirtual subsystem will reference model-wide header files, such as *model.h* and *model_private.h*.
- The **Function with separate data** option is incompatible with the **GRT compatible call interface** option, located on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box. Selecting both will generate an error.
- The **Function with separate data** option is incompatible with the **Generate reusable code** option (**Real-Time Workshop/Interface** pane). Selecting both will generate an error.
- Although the *model_initialize* function generated for a model containing a nonvirtual subsystem that uses the **Function with separate data** option may have a `firstTime` argument, the argument is not used. Thus, you cannot call *model_initialize* at a time greater than start time, for example, to reset block states. To suppress inclusion of the `firstTime` flag in the *model_initialize* function definition, set the model configuration parameter `IncludeERTFirstTime` to off.

Controlling Generation of Function Prototypes

- “Overview” on page 27-2
- “Configuring Model Function Prototypes” on page 27-4
- “Model Function Prototypes Example” on page 27-12
- “Configuring Model Function Prototypes Programmatically” on page 27-18
- “Sample Script for Configuring Model Function Prototypes” on page 27-22
- “Verifying Generated Code for Customized Functions” on page 27-23
- “Model Function Prototype Control Limitations” on page 27-24

Overview

The Real-Time Workshop Embedded Coder software provides a **Configure Model Functions** button, located on the **Interface** pane of the Configuration Parameters dialog box, that allows you to control the model function prototypes that are generated for ERT-based Simulink models.

By default, the function prototype of an ERT-based model's generated `model_step` function resembles the following:

```
void model_step(void);
```

The function prototype of an ERT-based model's generated `model_init` function resembles the following:

```
void model_init(void);
```

(For more detailed information about the default calling interface for the `model_step` function, see the `model_step` reference page.)

The **Configure Model Functions** button on the **Interface** pane provides you flexible control over the model function prototypes that are generated for your model. Clicking **Configure Model Functions** launches a Model Interface dialog box (see “Configuring Model Function Prototypes” on page 27-4). Based on the **Function specification** value you specify for your model function (supported values include `Default model initialize` and `step functions` and `Model specific C prototypes`), you can preview and modify the function prototypes. Once you validate and apply your changes, you can generate code based on your function prototype modifications.

For more information about using the **Configure Model Functions** button and the Model Interface dialog box, see “Model Function Prototypes Example” on page 27-12 and the demo model `rtwdemo_fcncnprotoctrl`, which is preconfigured to demonstrate function prototype control.

Alternatively, you can use function prototype control functions to programmatically control model function prototypes. For more information, see “Configuring Model Function Prototypes Programmatically” on page 27-18.

You can also control model function prototypes for nonvirtual subsystems, if you generate subsystem code using right-click build. To launch the **Model Interface for subsystem** dialog box, use the `RTW.configSubsystemBuild` function.

Right-click building the subsystem generates the step and initialization functions according to the customizations you make. For more information, see “Configuring Function Prototypes for Nonvirtual Subsystems” on page 27-9.

For limitations that apply, see “Model Function Prototype Control Limitations” on page 27-24.

Configuring Model Function Prototypes

In this section...

“Launching the Model Interface Dialog Boxes” on page 27-4

“Default Model Initialize and Step Functions View” on page 27-4

“Model Specific C Prototypes View” on page 27-5

“Configuring Function Prototypes for Nonvirtual Subsystems” on page 27-9

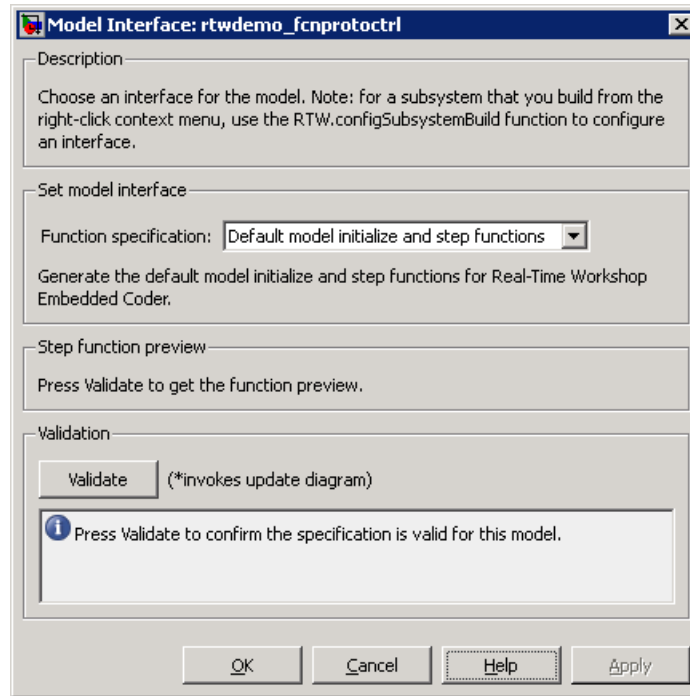
Launching the Model Interface Dialog Boxes

Clicking the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box launches the Model Interface dialog box. This dialog box is the starting point for configuring the model function prototypes that are generated during code generation for ERT-based Simulink models. Based on the **Function specification** value you select for your model function (supported values include `Default model initialize and step functions` and `Model specific C prototypes`), you can preview and modify the function prototype. Once you validate and apply your changes, you can generate code based on your function prototype modifications.

To configure function prototypes for a right-click build of a nonvirtual subsystem, invoke the `RTW.configSubsystemBuild` function, which launches the Model Interface for subsystem dialog box. For more information, see “Configuring Function Prototypes for Nonvirtual Subsystems” on page 27-9

Default Model Initialize and Step Functions View

The figure below shows the Model Interface dialog box in the `Default model initialize and step functions` view.



The **Default model initialize and step functions** view allows you to validate and preview the predicted default model step and initialization function prototypes. To validate the default function prototype configuration against your model, click the **Validate** button. If the validation succeeds, the predicted step function prototype appears in the **Step function preview** subpane.

Note You cannot use the **Default model initialize and step functions** view to modify the function prototype configuration.

Model Specific C Prototypes View

Selecting **Model specific C** prototypes for the **Function specification** parameter displays the **Model specific C** prototypes view of your model function prototypes. This view provides controls that you can use to customize

the function names, the order of arguments, and argument attributes including name, passing mechanism, and type qualifier for each of the model's root-level I/O ports.

To begin configuring your function control prototype configuration, click the **Get Default Configuration** button. This activates and initializes the function names and properties in the **Configure model initialize and step functions** subpane, as shown below. If you click **Get Default Configuration** again later, only the properties of the step function arguments are reset to default values.

Model Interface: rtdemo_fcprotoctrl

Description

Choose an interface for the model. Note: for a subsystem that you build from the right-click context menu, use the RTW.configSubsystemBuild function to configure an interface.

Set model interface

Function specification: **Model specific C prototypes**

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model initialize and step functions.

Get Default Configuration (*invokes update diagram)

Configure model initialize and step functions

Initialize function name: rtdemo_fcprotoctrl_init_custom

Step function name: rtdemo_fcprotoctrl_step_custom

Step function arguments:

Order	Port Name	Port Type	Category	Argument Name	Qualifier
Return	Out1	Outport	Value	Out1	none
1	In1	Inport	Value	argIn1	const
2	In2	Inport	Pointer	argIn2	const * const
3	Out2	Outport	Pointer	argOut2	none
4	In3	Inport	Pointer	argIn3	const * const
5	In4	Inport	Pointer	argIn4	none

Up

Down

Step function preview

Out1 = rtdemo_fcprotoctrl_step_custom (argIn1, * argIn2, * argOut2, * argIn3, * argIn4)

Validation

Validate (*invokes update diagram)

i Press Validate to confirm the specification is valid for this model.

OK Cancel Help Apply

In the **Configure model initialize and step functions** subpane:

Parameter	Description
Step function name	Name of the <i>model_step</i> function.
Initialize function name	Name of the <i>model_init</i> function.
Order	Order of the argument. A return argument is listed as Return .
Port Name	Name of the port.
Port Type	Type of the port.
Category	Specifies how an argument is passed in or out from the customized step function, either by copying a value (Value) or by a pointer to a memory space (Pointer).
Argument Name	Name of the argument.
Qualifier (optional)	<p>Specifies a const type qualifier for a function argument. The available values are dependent on the Category specified. When you change the Category, if the specified type is no longer available, the Qualifier changes to none. The possible values are:</p> <ul style="list-style-type: none"> • none • const (value) • const* (value referenced by the pointer) • const*const (value referenced by the pointer and the pointer itself)

Parameter	Description
	<p>Tip When a model includes a referenced model, the <code>const</code> type qualifier for the root input argument of the referenced model's specified step function interface is set to <code>none</code>, and the qualifier for the source signal in the referenced model's parent is set to a value other than <code>none</code>, Real-Time Workshop honors the referenced model's interface specification by generating a type cast that discards the <code>const</code> type qualifier from the source signal. To override this behavior, add a <code>const</code> type qualifier to the referenced model.</p>

The **Step function preview** subpane provides a preview of how your step function prototype is interpreted in generated code. The preview is updated dynamically as you make modifications.

An argument `foo` whose **Category** is `Pointer` is previewed as `* foo`. If its **Category** is `Value`, it is previewed as `foo`. Notice that argument types and qualifiers are not represented in the **Step function preview** subpane.

Configuring Function Prototypes for Nonvirtual Subsystems

You can control step and initialization function prototypes for nonvirtual subsystems in ERT-based Simulink models, if you generate subsystem code using right-click build. Function prototype control is supported for the following types of nonvirtual subsystems:

- Triggered subsystems
- Enabled subsystems
- Enabled trigger subsystems
- While subsystems

- For subsystems
- Stateflow subsystems if atomic
- Embedded MATLAB subsystems if atomic

To launch the Model Interface for Subsystem dialog box, open the model containing the subsystem and invoke the `RTW.configSubsystemBuild` function.

The Model Interface dialog box for modifying the model-specific C prototypes for the `rtwdemo_counter/Amplifier` subsystem appears as follows:

Model Interface for subsystem: Amplifier

Description

Choose an interface for the model. Note: for a subsystem that you build from the right-click context menu, use the RTW.configSubsystemBuild function to configure an interface.

Set model interface

Function specification: **Model specific C prototypes**

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model initialize and step functions.

Get Default Configuration (*invokes update diagram)

Configure model initialize and step functions

Initialize function name:

Step function name:

Step function arguments:

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	In	Inport	Value	arg_In	none
2	Trigger	Inport	Value	arg_Trigger	none
3	Out	Outport	Pointer	arg_Out	none

Up

Down

Step function preview

Amplifier_custom (arg_In, arg_Trigger, * arg_Out)

Validation

Validate (*invokes update diagram)

i Press Validate to confirm the specification is valid for this model.

OK Cancel Help Apply

Right-click building the subsystem generates the step and initialization functions according to the customizations you make.

Model Function Prototypes Example

The following procedure demonstrates how to use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box to control the model function prototypes that the Real-Time Workshop Embedded Coder software generates for your Simulink model.

- 1 Open a MATLAB session and launch the `rtwdemo_counter` demo model.
- 2 In the `rtwdemo_counter` Model Editor, double-click the **Generate Code Using Real-Time Workshop Embedded Coder (double-click)** button to generate code for an ERT-based version of `rtwdemo_counter`. The code generation report for `rtwdemo_counter` appears.
- 3 In the code generation report, click the link for `rtwdemo_counter.c`.
- 4 In the `rtwdemo_counter.c` code display, locate and examine the generated code for the `rtwdemo_counter_step` and the `rtwdemo_counter_initialize` functions:

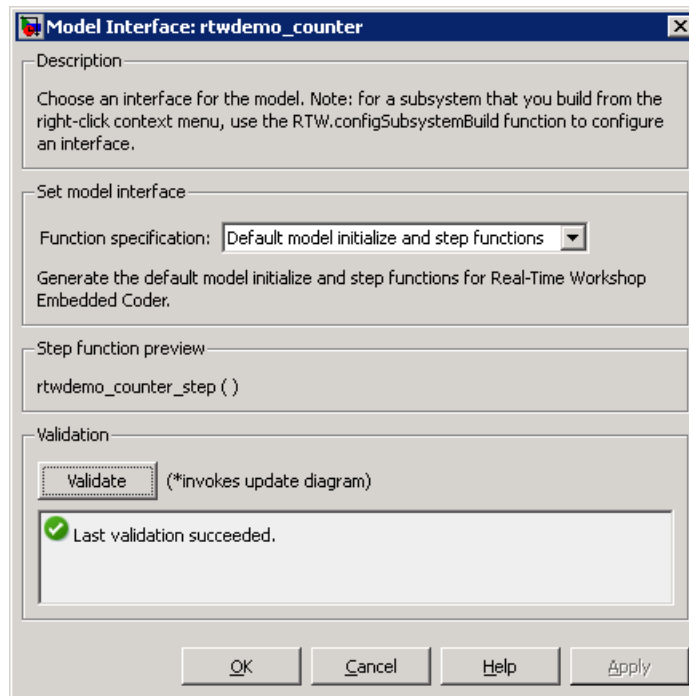
```
/* Model step function */
void rtwdemo_counter_step(void)
{
    ...
}

/* Model initialize function */
void rtwdemo_counter_initialize(void)
{
    ...
}
```

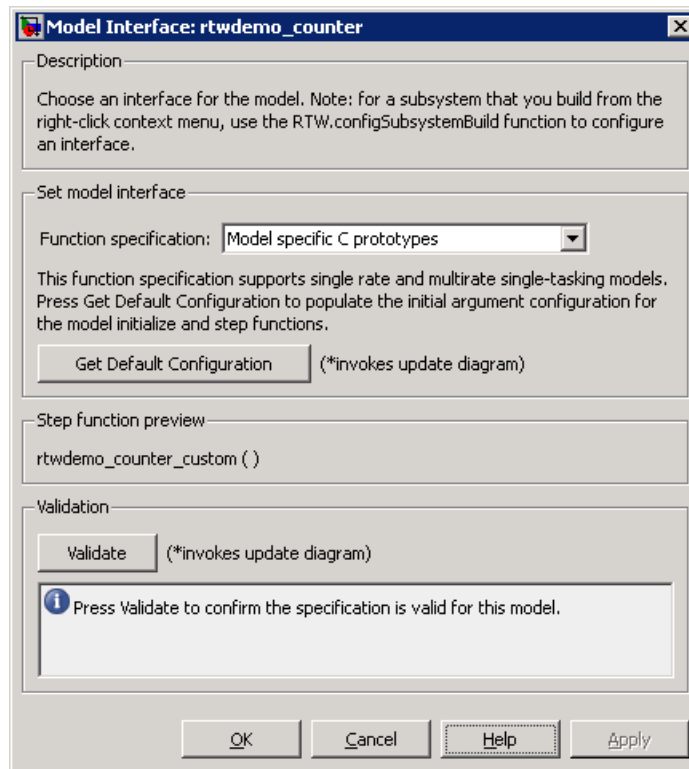
You can close the report window after you have examined the generated code. Optionally, you can save `rtwdemo_counter.c` and any other generated files of interest to a different location for later comparison.

- 5 From the `rtwdemo_counter` model, open the Configuration Parameters dialog box.

- 6 Navigate to the **Real-Time Workshop > Interface** pane and click the **Configure Model Functions** button. The Model Interface dialog box appears.
- 7 In the initial (Default model initialize and step functions) view of the Model Interface dialog box, click the **Validate** button to validate and preview the default function prototype for the `rtwdemo_counter_step` function. The function prototype arguments under **Step function preview** should correspond to the default prototype in step 4.



- 8 In the Model Interface dialog box, set **Function specification** field to Model specific C prototypes. Making this selection switches the dialog box from the Default model initialize and step functions view to the Model specific C prototypes view.



- 9 In the Model specific C prototypes view, click the **Get Default Configuration** button to activate the **Configure model initialize and step functions** subpane.

Model Interface: rtdemo_counter

Description

Choose an interface for the model. Note: for a subsystem that you build from the right-click context menu, use the RTW.configSubsystemBuild function to configure an interface.

Set model interface

Function specification: **Model specific C prototypes**

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model initialize and step functions.

Get Default Configuration (*invokes update diagram)

Configure model initialize and step functions

Initialize function name: rtdemo_counter_initialize

Step function name: rtdemo_counter_custom

Step function arguments:

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	Input	Inport	Value	arg_Input	none
2	Output	Outport	Pointer	arg_Output	none

Up

Down

Step function preview

rtdemo_counter_custom (arg_Input, * arg_Output)

Validation

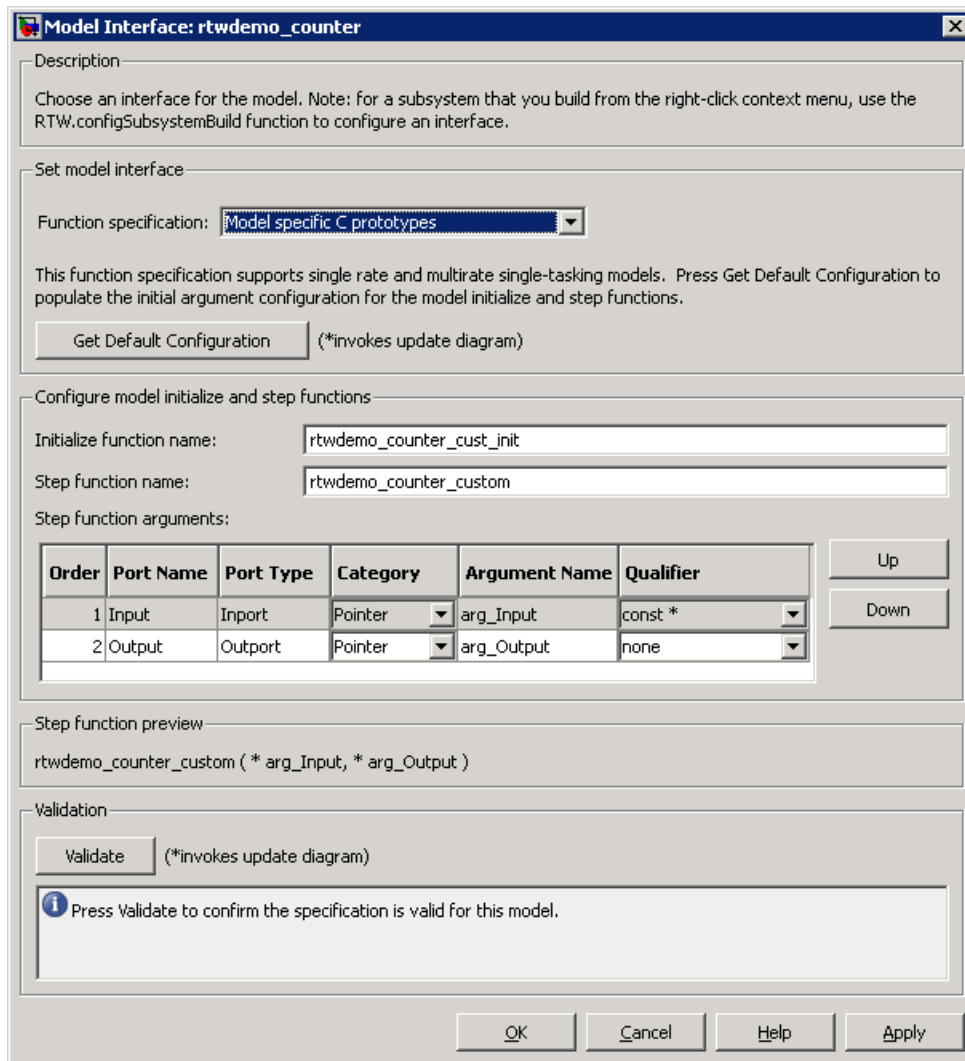
Validate (*invokes update diagram)

i Press Validate to confirm the specification is valid for this model.

OK Cancel Help Apply

- 10** In the **Configure model initialize and step functions** subpane, change **Initialize function name** to rtdemo_counter_cust_init.
- 11** In the **Configure model initialize and step functions** subpane, in the row for the Input argument, change the value of **Category** from Value to

Pointer and change the value of **Qualifier** from none to const *. The preview reflects your changes.



12 Click the **Validate** button to validate the modified function prototype. The **Validation** subpane displays a message that the validation succeeded.

- 13** Click **OK** to exit the Model Interface dialog box.
- 14** Generate code for the model. When the build completes, the code generation report for `rtwdemo_counter` appears.
- 15** In the code generation report, click the link for `rtwdemo_counter.c`.
- 16** Locate and examine the generated code for the `rtwdemo_counter_custom` and `rtwdemo_counter_cust_init` functions:

```
/* Customized model step function */
void rtwdemo_counter_custom(const int32_T *arg_Input, int32_T *arg_Output)
{
    ...
}

/* Model initialize function */
void rtwdemo_counter_cust_init(void)
{
    ...
}
```

- 17** Verify that the generated code is consistent with the function prototype modifications that you specified in the Model Interface dialog box.

Configuring Model Function Prototypes Programmatically

You can use the function prototype control functions (listed in Function Prototype Control Functions on page 27-20), to programmatically control model function prototypes. Typical uses of these functions include:

- **Create and validate a new function prototype**
 - 1 Create a model-specific C function prototype with `obj = RTW.ModelSpecificCPrototype`, where `obj` returns a handle to a newly created, empty function prototype.
 - 2 Add argument configuration information for your model ports using `RTW.ModelSpecificCPrototype.addArgConf`.
 - 3 Attach the function prototype to your loaded ERT-based Simulink model using `RTW.ModelSpecificCPrototype.attachToModel`.
 - 4 Validate the function prototype using `RTW.ModelSpecificCPrototype.runValidation`.
 - 5 If validation succeeds, save your model and then generate code using the `rtwbuild` function.
- **Modify and validate an existing function prototype**
 - 1 Get the handle to an existing model-specific C function prototype that is attached to your loaded ERT-based Simulink model with `obj = RTW.getFunctionSpecification(modelName)`, where `modelName` is a string specifying the name of a loaded ERT-based Simulink model, and `obj` returns a handle to a function prototype attached to the specified model.

You can use other function prototype control functions on the returned handle only if the test `isa(obj, 'RTW.ModelSpecificCPrototype')` returns 1. If the model does not have a function prototype configuration, the function returns []. If the function returns a handle to an object of type `RTW.FcnDefault`, you cannot modify the existing function prototype.

- 2 Use the `Get` and `Set` functions listed in Function Prototype Control Functions on page 27-20 to test and reset such items as the function names, argument names, argument positions, argument categories, and argument type qualifiers.

- 3** Validate the function prototype using `RTW.ModelSpecificCPrototype.runValidation`.
 - 4** If validation succeeds, save your model and then generate code using the `rtwbuild` function.
- **Create and validate a new function prototype, starting with default configuration information from your Simulink model**
 - 1** Create a model-specific C function prototype using `obj = RTW.ModelSpecificCPrototype`, where `obj` returns a handle to a newly created, empty function prototype.
 - 2** Attach the function prototype to your loaded ERT-based Simulink model using `RTW.ModelSpecificCPrototype.attachToModel`.
 - 3** Get default configuration information from your model using `RTW.ModelSpecificCPrototype.getDefaultConf`.
 - 4** Use the `Get` and `Set` functions listed in Function Prototype Control Functions on page 27-20 to test and reset such items as the function names, argument names, argument positions, argument categories, and argument type qualifiers.
 - 5** Validate the function prototype using `RTW.ModelSpecificCPrototype.runValidation`.
 - 6** If validation succeeds, save your model and then generate code using the `rtwbuild` function.

Note You should not use the same model-specific C function prototype object across multiple models. If you do, changes that you make to the step and initialization function prototypes in one model are propagated to other models, which is usually not desirable.

Function Prototype Control Functions

Function	Description
RTW.ModelSpecificCPrototype.addArgConf	Add step function argument configuration information for Simulink model port to model-specific C function prototype
RTW.ModelSpecificCPrototype.attachToModel	Attach model-specific C function prototype to loaded ERT-based Simulink model
RTW.ModelSpecificCPrototype.getArgCategory	Get step function argument category for Simulink model port from model-specific C function prototype
RTW.ModelSpecificCPrototype.getArgName	Get step function argument name for Simulink model port from model-specific C function prototype
RTW.ModelSpecificCPrototype.getArgPosition	Get step function argument position for Simulink model port from model-specific C function prototype
RTW.ModelSpecificCPrototype.getArgQualifier	Get step function argument type qualifier for Simulink model port from model-specific C function prototype
RTW.ModelSpecificCPrototype.getDefaultConf	Get default configuration information for model-specific C function prototype from Simulink model to which it is attached
RTW.ModelSpecificCPrototype.getFunctionName	Get function names from model-specific C function prototype
RTW.ModelSpecificCPrototype.getNumArgs	Get number of step function arguments from model-specific C function prototype
RTW.ModelSpecificCPrototype.getPreview	Get model-specific C function prototype code previews
RTW.configSubsystemBuild	Launch GUI to configure C function prototype or C++ encapsulation interface for right-click build of specified subsystem
RTW.getFunctionSpecification	Get handle to model-specific C function prototype object

Function Prototype Control Functions (Continued)

Function	Description
<code>RTW.ModelSpecificCPrototype.runValidation</code>	Validate model-specific C function prototype against Simulink model to which it is attached
<code>RTW.ModelSpecificCPrototype.setArgCategory</code>	Set step function argument category for Simulink model port in model-specific C function prototype
<code>RTW.ModelSpecificCPrototype.setArgName</code>	Set step function argument name for Simulink model port in model-specific C function prototype
<code>RTW.ModelSpecificCPrototype.setArgPosition</code>	Set step function argument position for Simulink model port in model-specific C function prototype
<code>RTW.ModelSpecificCPrototype.setArgQualifier</code>	Set step function argument type qualifier for Simulink model port in model-specific C function prototype
<code>RTW.ModelSpecificCPrototype.setFunctionName</code>	Set function names in model-specific C function prototype

Sample Script for Configuring Model Function Prototypes

The following sample MATLAB script configures the model function prototypes for the `rtwdemo_counter` model, using the Function Prototype Control Functions on page 27-20.

```
%% Open the rtwdemo_counter model
rtwdemo_counter

%% Select ert.tlc as the System Target File for the model
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Create a model-specific C function prototype
a=RTW.ModelSpecificCPrototype

%% Add argument configuration information for Input and Output ports
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

%% Attach the model-specific C function prototype to the model
attachToModel(a,gcs)

%% Rename the initialization function
setFunctionName(a,'InitFunction','init')

%% Rename the step function and change some argument attributes
setFunctionName(a,'StepFunction','step')
setArgPosition(a,'Output',1)
setArgCategory(a,'Input','Value')
setArgName(a,'Input','InputArg')
setArgQualifier(a,'Input','none')

%% Validate the function prototype against the model
[status,message]=runValidation(a)

%% if validation succeeded, generate code and build
if status
    rtwbuild(gcs)
end
```

Verifying Generated Code for Customized Functions

You can use software-in-the-loop (SIL) testing to verify the generated code for your customized step and initialization functions. This involves creating a SIL block with your generated code, which then can be integrated into a Simulink model to verify that the generated code provides the same result as the original model or nonvirtual subsystem. For more information, see Chapter 24, “Generating S-Function Wrappers” and Chapter 36, “Verifying Generated Source Code With Software-in-the-Loop Simulation”.

Model Function Prototype Control Limitations

The following limitations apply to controlling model function prototypes:

- Function prototype control supports only step and initialization functions generated from a Simulink model.
- Function prototype control supports only single-instance implementations. For standalone targets, you must clear the **Generate reusable code** check box (on the **Interface** pane of the Configuration Parameters dialog box). For model reference targets, you must select **One** for the **Total number of instances allowed per top model** parameter (on the **Model Referencing** pane of the Configuration Parameters dialog box).
- For model reference targets, the code generator ignores the **Generate reusable code** parameter (on the **Interface** pane of the Configuration Parameters dialog box).
- You must select the **Single output/update function** parameter (on the **Interface** pane of the Configuration Parameters dialog box).
- Function prototype control does not support multitasking models. Multirate models are supported, but you must configure the models for single-tasking.
- You must configure root-level inports and outports to use Auto storage classes.
- The generated code for a parent model does not call the function prototype control step functions generated from referenced models.
- Do not control function prototypes with the static `ert_main.c` provided by The MathWorks. Specifying a function prototype control configuration other than the default creates a mismatch between the generated code and the default static `ert_main.c`.
- The code generator removes the data structure for the root inports of the model unless a subsystem implemented by a nonreusable function uses the value of one or more of the inports.
- The code generator removes the data structure for the root outports of the model except when you enable MAT-file logging, or if the sample time of one or more of the outports is not the fundamental base rate (including a constant rate).

- If you copy a subsystem block and paste it to create a new block in either a new model or the same model, the function prototype control interface information from the original subsystem block does not copy to the new subsystem block.
- For a Stateflow chart that uses a model root inport value, or that calls a subsystem that uses a model root inport value, you must do one of the following to generate code:
 - Clear the **Execute (enter) Chart At Initialization** check box in the Stateflow chart.
 - Make the Stateflow function a nonreusable function.
 - Insert a Signal Conversion block immediately after the root inport and select the **Exclude this block from 'Block reduction' optimization** check box in the Signal Conversion block parameters.

Controlling Generation of Encapsulated C++ Model Interfaces

- “Overview of C++ Encapsulation” on page 28-2
- “C++ Encapsulation Quick-Start Example” on page 28-4
- “Generating and Configuring C++ Encapsulation Interfaces to Model Code” on page 28-12
- “Configuring C++ Encapsulation Interfaces Programmatically” on page 28-22
- “Sample Script for Configuring the Step Method for a Model Class” on page 28-25
- “C++ Encapsulation Interface Control Limitations” on page 28-27

Overview of C++ Encapsulation

Using the Real-Time Workshop Embedded Coder language option C++ (Encapsulated), you can generate a C++ class interface to model code. The generated interface encapsulates all required model data into C++ class attributes and all model entry point functions into C++ class methods. The benefits of encapsulation include:

- Greater control over access to model data
- Ability to multiply instantiate model classes
- Easier integration of model code into C++ programming environments

C++ encapsulation also works for right-click builds of nonvirtual subsystems. (For information on requirements that apply, see “Configuring C++ Encapsulation Interfaces for Nonvirtual Subsystems” on page 28-20.)

The general procedure for generating C++ encapsulation interfaces to model code is as follows:

- 1** Configure your model to use an `ert.tlc` system target file provided by The MathWorks.
- 2** Select the language option C++ (Encapsulated) for your model.
- 3** Optionally, configure related C++ encapsulation interface settings for your model code, using either a graphical user interface (GUI) or application programming interface (API).
- 4** Generate model code and examine the results.

To get started with an example, see “C++ Encapsulation Quick-Start Example” on page 28-4. For more details about configuring C++ encapsulation interfaces for your model code, see “Generating and Configuring C++ Encapsulation Interfaces to Model Code” on page 28-12 and “Configuring C++ Encapsulation Interfaces Programmatically” on page 28-22. For limitations that apply, see “C++ Encapsulation Interface Control Limitations” on page 28-27.

Note For a demonstration of the C++ encapsulation capability, see the demo model `rtwdemo_cppencap`.

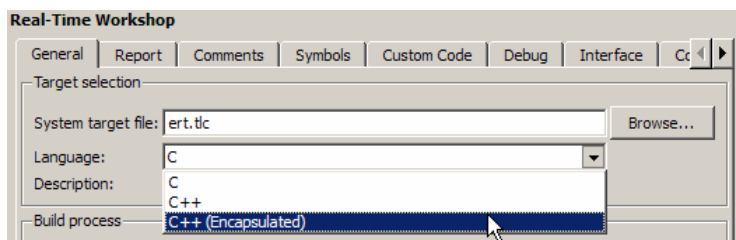
C++ Encapsulation Quick-Start Example

This example illustrates a simple use of the C++ (Encapsulated) option. It uses C++ encapsulation to generate interfaces for code from a Real-Time Workshop demo model, without extensive modifications to default settings.

Note For details about setting C++ encapsulation options, see the sections that follow this example, beginning with “Generating and Configuring C++ Encapsulation Interfaces to Model Code” on page 28-12.

To generate C++ encapsulated interfaces for a Simulink model:

- 1 Open a model for which you would like to generate C++ encapsulation interfaces. This example uses the Real-Time Workshop demo model `rtwdemo_counter`.
- 2 Configure the model to use an `ert.tlc` system target file provided by The MathWorks. For example, open the Configuration Parameters dialog box, go to the **Real-Time Workshop** pane, select an appropriate target value from the **System target file** menu, and click **Apply**.
- 3 Optionally, as a baseline for later code comparison, generate code from the model using a different **Language** parameter setting, C++ or C. (You can set up the build directory naming or location to distinguish your baseline build from later builds of the same model.)
- 4 On the **Real-Time Workshop** pane of the Configuration Parameters dialog box, select the C++ (Encapsulated) language option.



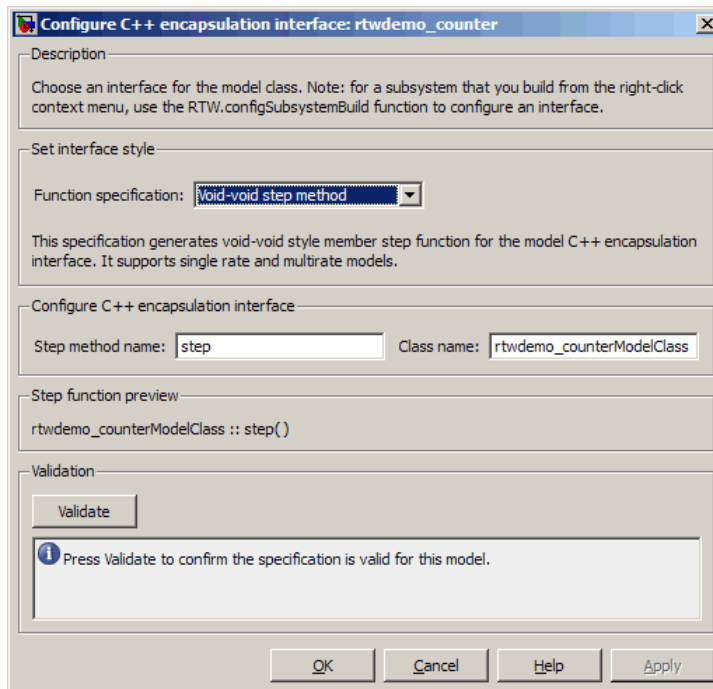
Click **Apply**.

Note To immediately generate the default style of encapsulated C++ code, without exploring the related model configuration options, skip steps 5–9 and go directly to step 10.

- 5 Go to the **Interface** pane of the Configuration Parameters dialog box and examine the **Code interface** subpane.

When you selected the C++ (Encapsulated) language option for your model, C++ encapsulation interface controls replaced the default options on the **Code interface** subpane. See “Configuring Code Interface Options” on page 28-13 for descriptions of these controls. Examine the default settings and modify as appropriate.

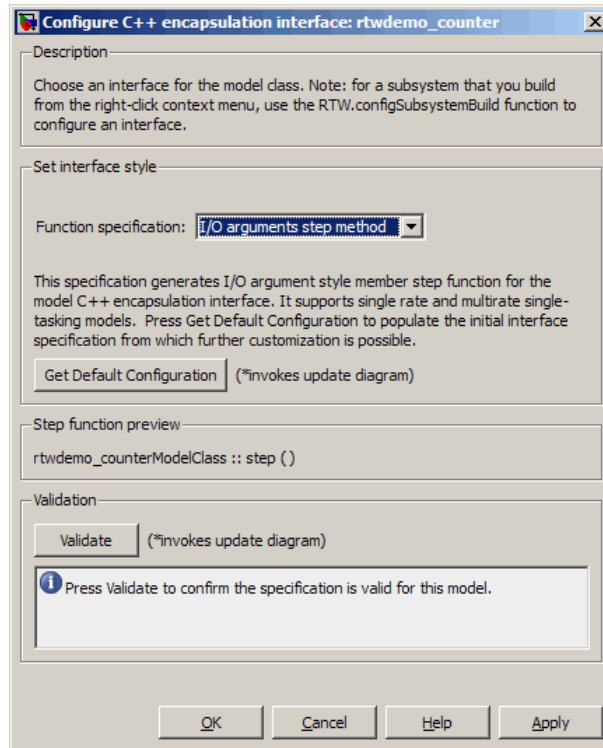
- 6 Click the **Configure C++ Encapsulation Interface** button. This action opens the Configure C++ encapsulation interface dialog box, which allows you to configure the step method for your generated model class. The dialog box initially displays a view for configuring a void-void style step method (passing no I/O arguments) for the model class. In this view, you can rename the model class and the step method for your model.



See “Configuring the Step Method for Your Model Class” on page 28-15 for descriptions of these controls.

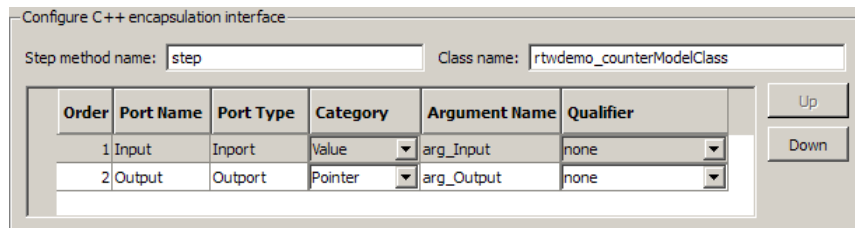
Note If the void-void interface style meets your needs, you can skip steps 7–9 and go directly to step 10.

- 7 If you want root-level model input and output to be arguments on the step method, select the value **I/O arguments step method** from the **Function specification** menu. The dialog box displays a view for configuring an I/O arguments style step method for the model class.



See “Configuring the Step Method for Your Model Class” on page 28-15 for descriptions of these controls.

- 8 Click the **Get Default Configuration** button. This action causes a **Configure C++ encapsulation interface** subpane to appear in the dialog box. The subpane displays the initial interface configuration for your model, which provides a starting point for further customization.



See “Passing I/O Arguments” on page 28-17 for descriptions of these controls.

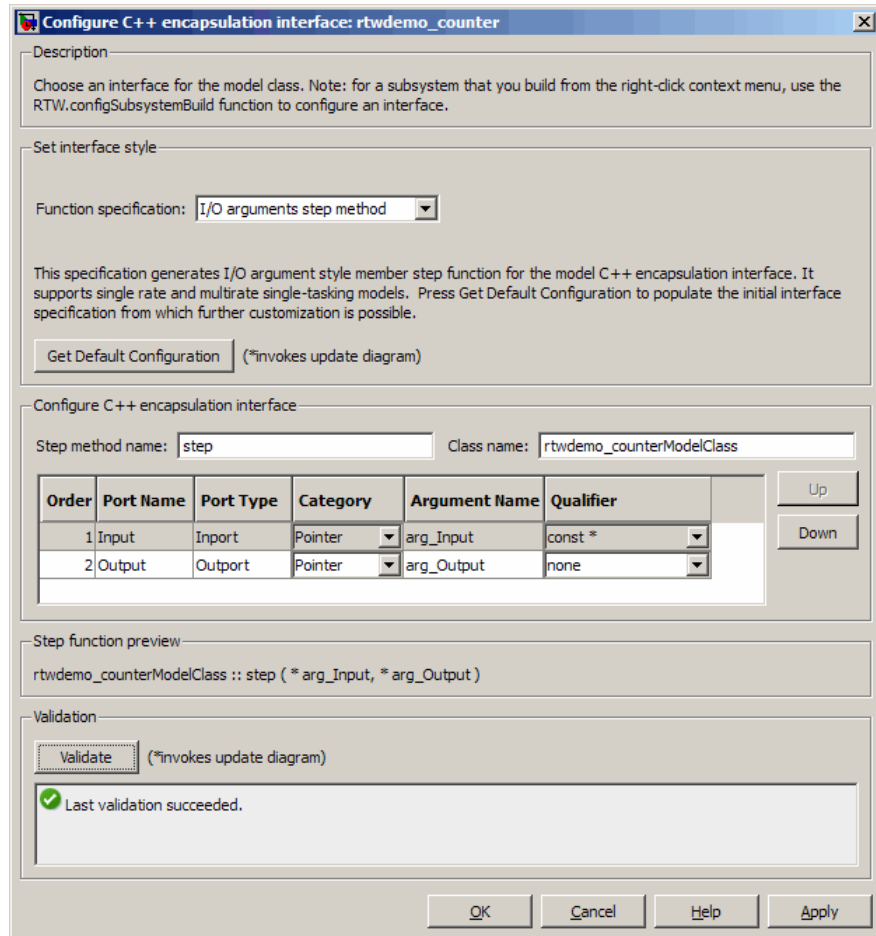
- 9 Perform this optional step only if you want to customize the configuration of the I/O arguments generated for your model step method.

Note If you choose to skip this step, you should click **Cancel** to exit the dialog box.

If you choose to perform this step, first you must check that the required option **Remove root level I/O zero initialization** is selected on the **Optimization** pane, and then navigate back to the I/O arguments step method view of the Configure C++ encapsulation interface dialog box.

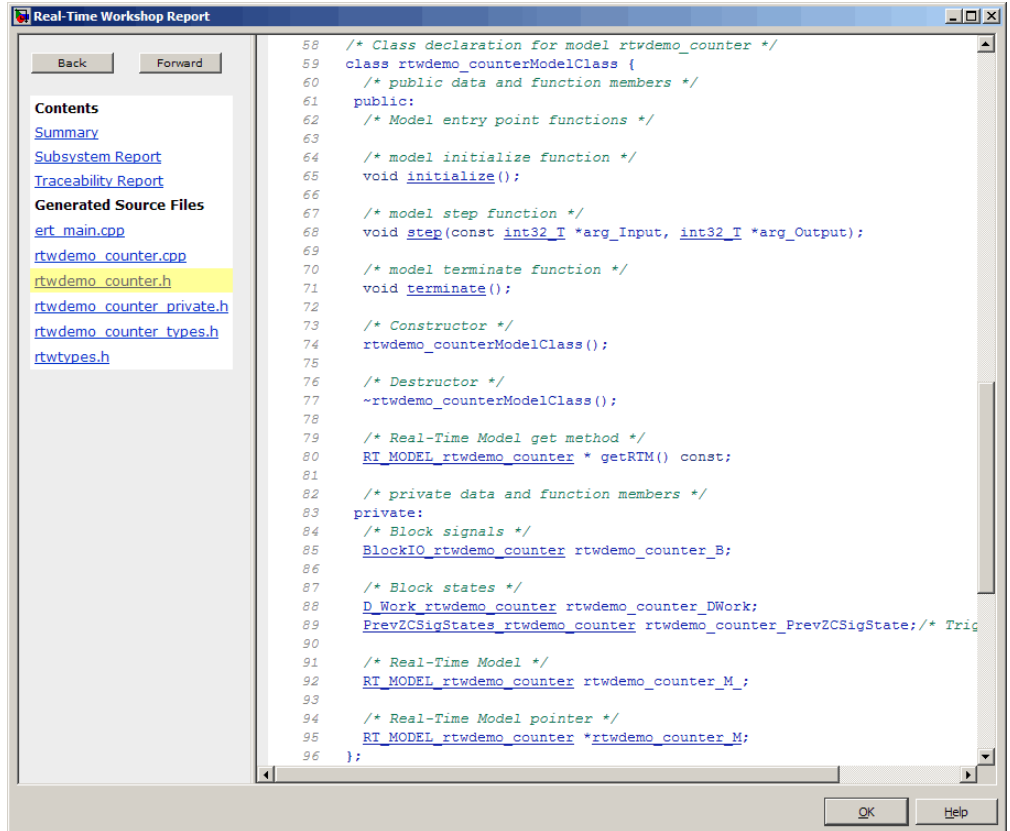
Now you can use the dialog box controls to configure I/O argument attributes. For example, in the **Configure C++ encapsulation interface** subpane, in the row for the Input argument, you can change the value of **Category** from Value to Pointer and change the value of **Qualifier** from none to const *. The preview updates to reflect your changes. Click the **Validate** button to validate the modified interface configuration.

Continue modifying and validating until you are satisfied with the step method configuration.



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

- 10 Generate code for the model. When the build completes, the code generation report for `rtwdemo_counter` appears. Examine the report and observe that all required model data is encapsulated into C++ class attributes and all model entry point functions are encapsulated into C++ class methods. For example, click the link for `rtwdemo_counter.h` to see the class declaration for the model.



The screenshot shows a window titled "Real-Time Workshop Report". On the left side, there is a navigation pane with a "Contents" section containing links for "Summary", "Subsystem Report", and "Traceability Report". Below that is a "Generated Source Files" section with links for "ert_main.cpp", "rtwdemo_counter.cpp", "rtwdemo_counter.h" (highlighted in yellow), "rtwdemo_counter_private.h", "rtwdemo_counter_types.h", and "rtwtypes.h". The main area of the window displays C++ code for the `rtwdemo_counter` class, with line numbers 58 through 96 on the left margin. The code includes comments and declarations for public and private members, as well as Real-Time Model (RT_MODEL) related methods and pointers.

```
58  /* Class declaration for model rtwdemo_counter */
59  class rtwdemo_counterModelClass {
60  /* public data and function members */
61  public:
62  /* Model entry point functions */
63
64  /* model initialize function */
65  void initialize();
66
67  /* model step function */
68  void step(const int32_I *arg_Input, int32_I *arg_Output);
69
70  /* model terminate function */
71  void terminate();
72
73  /* Constructor */
74  rtwdemo_counterModelClass();
75
76  /* Destructor */
77  ~rtwdemo_counterModelClass();
78
79  /* Real-Time Model get method */
80  RT_MODEL_rtwdemo_counter * getRTM() const;
81
82  /* private data and function members */
83  private:
84  /* Block signals */
85  BlockIO_rtwdemo_counter rtwdemo_counter_B;
86
87  /* Block states */
88  D_Work_rtwdemo_counter rtwdemo_counter_DWork;
89  PrevZCSigStates_rtwdemo_counter rtwdemo_counter_PrevZCSigState; /* Trig
90
91  /* Real-Time Model */
92  RT_MODEL_rtwdemo_counter rtwdemo_counter_M ;
93
94  /* Real-Time Model pointer */
95  RT_MODEL_rtwdemo_counter *rtwdemo_counter_M;
96  };
```

Note

- If you configured custom I/O arguments for the model step method (optional step 9), examine the generated code for the step method in `rtwdemo_counter.h` and `rtwdemo_counter.cpp`. The arguments should reflect your changes. For example, if you performed the Input argument modifications in step 9, the input argument should appear as `const int32_T *arg_Input`.
 - If you saved a baseline model build (optional step 3), you can traverse and compare the generated files in the corresponding build directories.
-

Generating and Configuring C++ Encapsulation Interfaces to Model Code

In this section...

“Selecting the C++ (Encapsulated) Option” on page 28-12

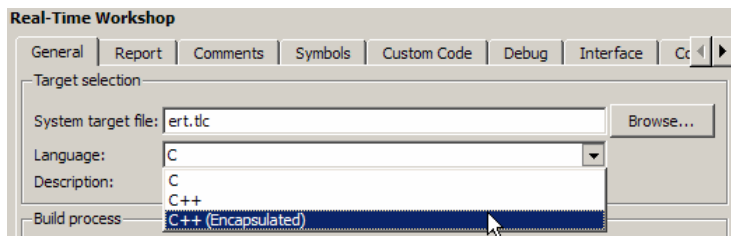
“Configuring Code Interface Options” on page 28-13

“Configuring the Step Method for Your Model Class” on page 28-15

“Configuring C++ Encapsulation Interfaces for Nonvirtual Subsystems” on page 28-20

Selecting the C++ (Encapsulated) Option

To select the C++ (Encapsulated) option, in the Configuration Parameters dialog box, on the **Real-Time Workshop** pane, use the **Language** menu:



When you select this option, you see the following effects on other panes in the Configuration Parameters dialog box:

- Disables model configuration options that C++ (Encapsulated) does not support. For details, see “C++ Encapsulation Interface Control Limitations” on page 28-27.
- Replaces the default options on the **Interface** pane, in the **Code interface** subpane, with C++ encapsulation interface controls, which are described in the next section.

Configuring Code Interface Options

When you select the C++ (Encapsulated) option for your model, the C++ encapsulation interface controls shown below replace the **Code interface** default options on the **Interface** pane.

- **Block parameter visibility**

Specifies whether to generate the block parameter structure as a public, private, or protected data member of the C++ model class (private by default).

- **Internal data visibility**

Specifies whether to generate internal data structures, such as Block I/O, DWork vectors, Runtime model, Zero-crossings, and continuous states, as public, private, or protected data members of the C++ model class (private by default).

- **Block parameter access**

Specifies whether to generate access methods for block parameters for the C++ model class (None by default). You can select noninlined access methods (Method) or inlined access methods (Inlined method).

- **Internal data access**

Specifies whether to generate access methods for internal data structures, such as Block I/O, DWork vectors, Runtime model, Zero-crossings, and continuous states, for the C++ model class (None by default). You can select noninlined access methods (Method) or inlined access methods (Inlined method).

- **External I/O access**

Specifies whether to generate access methods for root-level I/O signals for the C++ model class (**None** by default). You can select noninlined access methods (**Method**) or inlined access methods (**Inlined method**).

Note This parameter affects generated code only if you are using the default (**void-void style**) step method for your model class; *not* if you are explicitly passing arguments for root-level I/O signals using an **I/O arguments style** step method. For more information, see “Passing No Arguments (**void-void**)” on page 28-16 and “Passing I/O Arguments” on page 28-17.

- **Terminate function**

Specifies whether to generate the `model_terminate` function (on by default). This function contains all model termination code and should be called as part of system shutdown.

- **Generate destructor**

Specifies whether to generate a destructor for the C++ model class (on by default).

- **Use operator new for referenced model object registration**

For a model containing Model blocks, specifies whether generated code should use dynamic memory allocation, during model object registration, to instantiate objects for referenced models configured with a C++ encapsulation interface (off by default). If you select this option, during instantiation of an object for the top model in a model reference hierarchy, the generated code uses the operator `new` to instantiate objects for referenced models.

Selecting this option frees a parent model from having to maintain information about submodels beyond its direct children. Clearing this option means that a parent model maintains information about all of its submodels, including its direct and indirect children.

Note If you select this option, be aware that a `bad_alloc` exception might be thrown, per the C++ standard, if an out-of-memory error occurs during the use of `new`. You must provide code to catch and process the `bad_alloc` exception in case an out-of-memory error occurs for a `new` call during construction of a top model object.

- **Generate preprocessor conditionals**

For a model containing Model blocks, specifies whether to generate preprocessor conditional directives globally for a model, locally for each variant Model block, or conditionally based on the **Generate preprocessor conditionals** setting in the Model Reference Parameter dialog for each variant Model block (Use `local settings` by default).

- **Suppress error status in real-time model data structure**

Specifies whether to omit the error status field from the generated real-time model data structure `rtModel` (off by default). Selecting this option reduces memory usage.

Be aware that selecting this option can cause the code generator to omit the `rtModel` data structure from generated code.

- **Configure C++ Encapsulation Interface**

Opens the Configure C++ encapsulation interface dialog box, which allows you to configure the step method for your model class. For more information, see “Configuring the Step Method for Your Model Class” on page 28-15.

Configuring the Step Method for Your Model Class

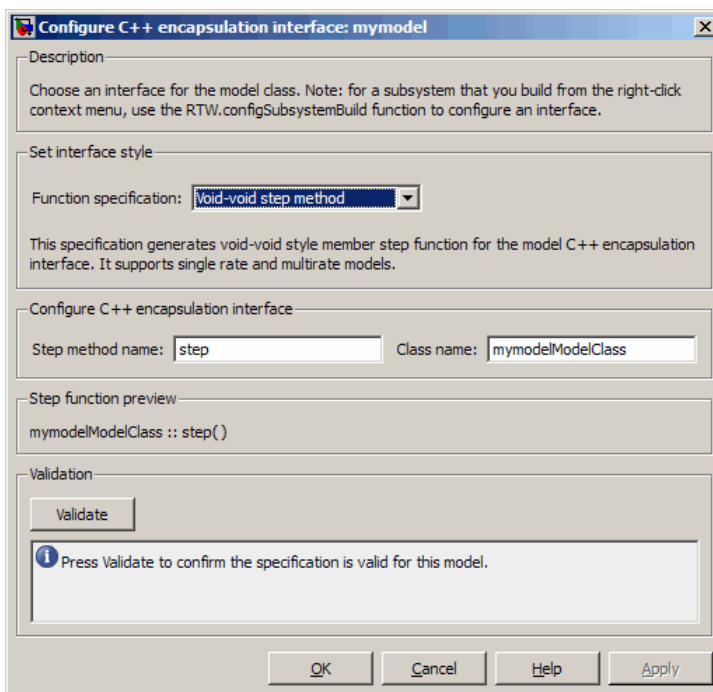
To configure the step method for your model class, on the **Interface** pane, click the **Configure C++ Encapsulation Interface** button, which is available when you select `C++ (Encapsulated)` for your model. This action opens the Configure C++ encapsulation interface dialog box, where you can configure the step method for your model class in either of two styles:

- “Passing No Arguments (void-void)” on page 28-16
- “Passing I/O Arguments” on page 28-17

Note The void-void style of step method specification supports single-rate models and multirate models, while the I/O arguments style supports single-rate models and multirate single-tasking models.

Passing No Arguments (void-void)

The Configure C++ encapsulation interface dialog box initially displays a view for configuring a void-void style step method for the model class.



- **Step method name**

Allows you to specify a step method name other than the default, `step`.

- **Class name**

Allows you to specify a model class name other than the default, `modelModelClass`.

- **Step function preview**

Displays a preview of the model step function prototype as currently configured. The preview display is dynamically updated as you make configuration changes.

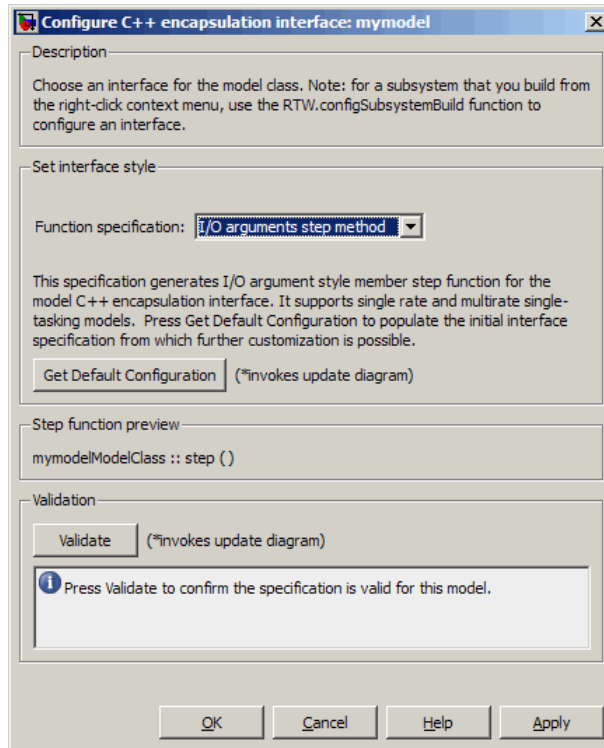
- **Validate**

Validates your current model step function configuration. The **Validation** pane displays success or failure status and an explanation of any failure.

Passing I/O Arguments

If you select **I/O arguments** step method from the **Function specification** menu, the dialog box displays a view for configuring an **I/O arguments** style step method for the model class.

Note To use the **I/O arguments** style step method, you must select the option **Remove root level I/O zero initialization** on the **Optimization** pane of the Configuration Parameters dialog box.



- **Get Default Configuration**

Click this button to get the initial interface configuration that provides a starting point for further customization.

- **Step function preview**

Displays a preview of the model step function prototype as currently configured. The preview dynamically updates as you make configuration changes.

- **Validate**

Validates your current model step function configuration. The **Validation** pane displays success or failure status and an explanation of any failure.

When you click **Get Default Configuration**, the **Configure C++ encapsulation interface** subpane appears in the dialog box, displaying the initial interface configuration. For example:

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	In1	Inport	Value	arg_In1	none
2	In2	Inport	Value	arg_In2	none
3	In3	Inport	Value	arg_In3	none
4	Out1	Outport	Pointer	arg_Out1	none
5	Out2	Outport	Pointer	arg_Out2	none

- **Step method name**

Allows you to specify a step method name other than the default, `step`.

- **Class name**

Allows you to specify a model class name other than the default, `modelModelClass`.

- **Order**

Displays the numerical position of each argument. Use the **Up** and **Down** buttons to change argument order.

- **Port Name**

Displays the port name of each argument (not configurable using this dialog box).

- **Port Type**

Displays the port type, Inport or Outport, of each argument (not configurable using this dialog box).

- **Category**

Displays the passing mechanism for each argument. To change the passing mechanism for an argument, select Value, Pointer, or Reference from the argument's **Category** menu.

- **Argument Name**

Displays the name of each argument. To change an argument name, click in the argument's **Argument name** field, position the cursor for text entry, and enter the new name.

- **Qualifier**

Displays the const type qualifier for each argument. To change the qualifier for an argument, select an available value from the argument's **Qualifier** menu. The possible values are:

- none
- const (value)
- const* (value referenced by the pointer)
- const*const (value referenced by the pointer and the pointer itself)
- const & (value referenced by the reference)

Tip When a model includes a referenced model, the const type qualifier for the root input argument of the referenced model's specified step function interface is set to **none** and the qualifier for the source signal in the referenced model's parent is set to a value other than **none**, Real-Time Workshop honors the referenced model's interface specification by generating a type cast that discards the const type qualifier from the source signal. To override this behavior, add a const type qualifier to the referenced model.

Configuring C++ Encapsulation Interfaces for Nonvirtual Subsystems

C++ encapsulation interfaces can be configured for right-click builds of nonvirtual subsystems in Simulink models, provided that:

- You select the system target file `ert.tlc` for the model.
- You select the **Language** parameter value **C++ (Encapsulated)** for the model.
- The subsystem is convertible to a Model block using the function `Simulink.SubSystem.convertToModelReference`. For referenced model conversion requirements, see the Simulink reference page `Simulink.SubSystem.convertToModelReference`.

To configure C++ encapsulation interfaces for a subsystem that meets the requirements:

- 1 Open the containing model and select the subsystem block.
- 2 Enter the following MATLAB command:

```
RTW.configSubsystemBuild(gcb)
```

where `gcb` is the Simulink function `gcb`, returning the full block path name of the current block.

This command opens a subsystem equivalent of the Configure C++ encapsulation interface dialog sequence that is described in detail in the preceding section, “Configuring the Step Method for Your Model Class” on page 28-15. (For more information about using the MATLAB command, see `RTW.configSubsystemBuild`.)

- 3 Use the Configure C++ encapsulation interface dialog boxes to configure C++ encapsulation settings for the subsystem.
- 4 Right-click the subsystem and select **Real-Time Workshop > Build Subsystem**.
- 5 When the subsystem build completes, you can examine the C++ encapsulation interfaces in the generated files and the HTML code generation report.

Configuring C++ Encapsulation Interfaces Programmatically

If you select the **Language** option C++ (Encapsulated) for your model, you can use the C++ encapsulation interface control functions (listed in C++ Encapsulation Interface Control Functions on page 28-23) to programmatically configure the step method for your model class.

Typical uses of these functions include:

- **Create and validate a new step method interface, starting with default configuration information from your Simulink model**
 - 1 Create a model-specific C++ encapsulation interface with *obj* = `RTW.ModelCPPVoidClass` or *obj* = `RTW.ModelCPPArgsClass`, where *obj* returns a handle to an newly created, empty C++ encapsulation interface.
 - 2 Attach the C++ encapsulation interface to your loaded ERT-based Simulink model using `attachToModel`.
 - 3 Get default C++ encapsulation interface configuration information from your model using `getDefaultConf`.
 - 4 Use the `Get` and `Set` functions listed in C++ Encapsulation Interface Control Functions on page 28-23 to test or reset the model class name and model step method name. Additionally, if you are using the I/O arguments style step method, you can test and reset argument names, argument positions, argument categories, and argument type qualifiers.
 - 5 Validate the C++ encapsulation interface using `runValidation`. (If validation fails, use the error message information that `runValidation` returns to address the issues.)
 - 6 Save your model and then generate code using the `rtwbuild` function.
- **Modify and validate an existing step method interface for a Simulink model**
 - 1 Get the handle to an existing model-specific C++ encapsulation interface that is attached to your loaded ERT-based Simulink model using *obj* = `RTW.getEncapsulationInterfaceSpecification(modelName)`, where *modelName* is a string specifying the name of a loaded ERT-based

Simulink model, and *obj* returns a handle to a C++ encapsulation interface attached to the specified model. If the model does not have an attached C++ encapsulation interface configuration, the function returns [].

- 2 Use the **Get** and **Set** functions listed in C++ Encapsulation Interface Control Functions on page 28-23 to test or reset the model class name and model step method name. Additionally, if the returned interface uses the I/O arguments style step method, you can test and reset argument names, argument positions, argument categories, and argument type qualifiers.
- 3 Validate the C++ encapsulation interface using `runValidation`. (If validation fails, use the error message information that `runValidation` returns to address the issues.)
- 4 Save your model and then generate code using the `rtwbuild` function.

Note You should not use the same model-specific C++ encapsulation interface control object across multiple models. If you do, changes that you make to the step method configuration in one model propagate to other models, which is usually not desirable.

C++ Encapsulation Interface Control Functions

Function	Description
<code>attachToModel</code>	Attach model-specific C++ encapsulation interface to loaded ERT-based Simulink model
<code>getArgCategory</code>	Get argument category for Simulink model port from model-specific C++ encapsulation interface
<code>getArgName</code>	Get argument name for Simulink model port from model-specific C++ encapsulation interface
<code>getArgPosition</code>	Get argument position for Simulink model port from model-specific C++ encapsulation interface
<code>getArgQualifier</code>	Get argument type qualifier for Simulink model port from model-specific C++ encapsulation interface

C++ Encapsulation Interface Control Functions (Continued)

Function	Description
getClassName	Get class name from model-specific C++ encapsulation interface
getDefaultConf	Get default configuration information for model-specific C++ encapsulation interface from Simulink model to which it is attached
getNumArgs	Get number of step method arguments from model-specific C++ encapsulation interface
getStepMethodName	Get step method name from model-specific C++ encapsulation interface
RTW.configSubsystemBuild	Open GUI to configure C function prototype or C++ encapsulation interface for right-click build of specified subsystem
RTW.getEncapsulation-InterfaceSpecification	Get handle to model-specific C++ encapsulation interface control object
runValidation	Validate model-specific C++ encapsulation interface against Simulink model to which it is attached
setArgCategory	Set argument category for Simulink model port in model-specific C++ encapsulation interface
setArgName	Set argument name for Simulink model port in model-specific C++ encapsulation interface
setArgPosition	Set argument position for Simulink model port in model-specific C++ encapsulation interface
setArgQualifier	Set argument type qualifier for Simulink model port in model-specific C++ encapsulation interface
setClassName	Set class name in model-specific C++ encapsulation interface
setStepMethodName	Set step method name in model-specific C++ encapsulation interface

Sample Script for Configuring the Step Method for a Model Class

The following sample MATLAB script configures the step method for the `rtwdemo_counter` model class, using the C++ Encapsulation Interface Control Functions on page 28-23.

```
%% Open the rtwdemo_counter model
rtwdemo_counter

%% Select ert.tlc as the System Target File for the model
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Select C++ (Encapsulated) as the target language for the model
set_param(gcs,'TargetLang','C++ (Encapsulated)')

%% Set required option for I/O arguments style step method (cmd off = GUI on)
set_param(gcs,'ZeroExternalMemoryAtStartup','off')

%% Create a C++ encapsulated interface using an I/O arguments style step method
a=RTW.ModelCPPArgsClass

%% Attach the C++ encapsulated interface to the model
attachToModel(a,gcs)

%% Get the default C++ encapsulation interface configuration from the model
getDefaultConf(a)

%% Move the Output port argument from position 2 to position 1
setArgPosition(a,'Output',1)

%% Reset the model step method name from step to StepMethod
setStepMethodName(a,'StepMethod')

%% Change the Input port argument name, category, and qualifier
setArgName(a,'Input','inputArg')
setArgCategory(a,'Input','Pointer')
setArgQualifier(a,'Input','const *')
```

```
%% Validate the function prototype against the model
[status,message]=runValidation(a)

%% if validation succeeded, generate code and build
if status
    rtwbuild(gcs)
end
```

C++ Encapsulation Interface Control Limitations

- The C++ (Encapsulated) option does not support some Simulink model configuration options. Selecting C++ (Encapsulated) disables the following items in the Configuration Parameters dialog box:
 - **Identifier format control** subpane on the **Symbols** pane
 - **Templates** pane
 - The **Templates** pane parameter **File customization template** is not supported for C++ (Encapsulated) code generation.
 - Selecting C++ (Encapsulated) turns on the **Templates** pane option **Generate an example main program** but removes it from the Configuration Parameters dialog box. If desired, you can disable it using the command line parameter `GenerateSampleERTMain`.
 - **Data Placement** pane
 - **Memory Sections** pane

Note Selecting C++ (Encapsulated) forces on the **Real-Time Workshop** pane model option **Ignore custom storage classes**. By design, C++ (Encapsulated) code generation treats data objects with custom storage classes as if their storage class attribute is set to `Auto`.

- Among the data exchange interfaces available on the **Interface** pane of the Configuration Parameters dialog box, only the `C API` interface is supported for C++ (Encapsulated) code generation. If you select `External mode` or `ASAP2`, code generation fails with a validation error.
- The I/O arguments style of step method specification supports single-rate models and multirate single-tasking models, but not multirate multitasking models.
- The C++ (Encapsulated) option does not support use of the `IncludeERTFirstTime` model option to include the `firstTime` argument in the `model_initialize` function generated for an ERT-based models. (The `IncludeERTFirstTime` option is off by default except for models created with R2006a.) Also, the C++ (Encapsulated) option requires that the target selected for the model support `firstTime` argument control by setting

the `ERTFirstTimeCompliant` target option, which all targets provided by The MathWorks do by default. In other words, the C++ (Encapsulated) option requires that the target option `ERTFirstTimeCompliant` is on and the model option `IncludeERTFirstTime` is off.

- The **Real-Time Workshop > Export Functions** capability does not support C++ (Encapsulated) as the target language.
- For a Stateflow chart that resides in a root model configured to use the I/O arguments `step` method function specification, and that uses a model root inport value or calls a subsystem that uses a model root inport value, you must do one of the following to generate code:
 - Clear the **Execute (enter) Chart At Initialization** check box in the Stateflow chart.
 - Insert a Signal Conversion block immediately after the root inport and select the **Exclude this block from 'Block reduction' optimization** check box in the Signal Conversion block parameters.
- When building a referenced model that is configured to generate a C++ encapsulation interface:
 - You must use the I/O arguments `step` method style of the C++ encapsulated interface. The `void-void` `step` method style is not supported for referenced models.
 - You cannot use a C++ encapsulation interface in cases when a referenced model cannot have a combined output/update function. Cases include a model that
 - Has multiple sample times
 - Has a continuous sample time
 - Saves states

Replacing Math Functions and Operators Using Target Function Libraries

- “Introduction to Target Function Libraries” on page 29-2
- “Creating Function Replacement Tables” on page 29-16
- “Examining and Validating Function Replacement Tables” on page 29-140
- “Registering Target Function Libraries” on page 29-149
- “Target Function Library Limitations” on page 29-158

Introduction to Target Function Libraries

In this section...

“Overview of Target Function Libraries” on page 29-2

“Target Function Libraries General Workflow” on page 29-7

“Target Function Libraries Quick-Start Example” on page 29-9

Overview of Target Function Libraries

The Real-Time Workshop Embedded Coder software provides the target function library (TFL) API, which allows you to create and register function replacement tables. When selected for a model, these TFL tables provide the basis for replacing default math functions and operators in your model code with target-specific code. The ability to control function and operator replacements potentially allows you to optimize target performance (speed and memory) and better integrate model code with external and legacy code.

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 software provides default TFLs, described in the following table. You select these TFLs 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

TFL	Description	Contains Tables...
GNU99 (GNU)	Generates calls to the GNU ⁷ 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

When a TFL contains multiple tables, the order in which they are listed reflects the order in which they are searched. The TFL API allows you to create your own TFLs, made up of your own function tables in combination with one of the default TFLs. For example, you could create a TFL for an embedded processor that combines some special-purpose function customizations with a processor-specific library of function and operator implementations:

MyProcessor (ANSI)	Generates calls to my custom function implementations or a processor-specific library.	tfl_table_sincfns.m tfl_table_myprocessor.m ansi_tfl_table_tmw.mat
--------------------	--	--

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')

7. GNU[®] is a registered trademark of the Free Software Foundation.

- A set of conceptual arguments that observe a Simulink naming scheme ('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 you require 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 your implementation function (such as 'cos_db1' or 'u8_add_u8_u8')
- A set of implementation arguments that you define (the order of which must correspond to the conceptual arguments), along with their I/O types (output or input) and data types
- Parameters providing the build information for your implementation function, including header file and source file names and paths

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. 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 (by `RTW.viewTFL` or by 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 is used over a similar entry with a lower priority (higher number).

The TFL API supports the following functions for replacement with custom library functions using TFL tables:

Math Functions			
Note For detailed support information, see “Example: Mapping Math Functions to Target-Specific Implementations” on page 29-27.			
abs	cos	log	saturate
acos	cosh	log10	sign
acosh	exactrSqrt	max	sin
asin	exp	min	sinh
asinh	fix	mod/fmod	sqrt
atan	floor	pow	tan
atan2	hypot	rem	tanh
atanh	ldexp	round	
ceil	ln	rSqrt	
Copy Utility Function			
memcpy			
Nonfinite Support Utility Functions			
getInf	getMinusInf	getNaN	

The TFL API also supports the following operations for replacement with custom library functions using TFL tables:

Note Unless otherwise stated, the listed operators are supported for the following input data types:

- single, double, and their complex equivalents
- int8, int16, int32, and their complex equivalents
- uint8, uint16, uint32, and their complex equivalents
- Fixed-point data types
- Mixed data types (different types for different inputs)

Operator	Key	Scalar Inputs	Nonscalar Inputs
Addition (+)	RTW_OP_ADD	Yes	Yes
Subtraction (-)	RTW_OP_MINUS	Yes	Yes
Multiplication (*, .*)	RTW_OP_MUL	Yes	Yes, including the ability to map to Basic Linear Algebra Subroutine (BLAS) multiplication functions
Division (/)	RTW_OP_DIV	Yes	—
Data type conversion (cast)	RTW_OP_CAST	Yes	—
Shift left (<<)	RTW_OP_SL	Yes, for integer and fixed-point data types	—

Operator	Key	Scalar Inputs	Nonscalar Inputs
Shift right (>>)	RTW_OP_SRA (arithmetic) ⁸ RTW_OP_SRL (logical)	Yes, for integer and fixed-point data types	—
Complex conjugation	RTW_OP_CONJUGATE	Yes	Yes
Transposition (.')	RTW_OP_TRANS	—	Yes
Hermitian (complex conjugate) transposition (')	RTW_OP_HERMITIAN	—	Yes
Multiplication with transposition	RTW_OP_TRMUL	—	Yes, including the ability to map to BLAS multiplication functions
Multiplication with Hermitian transposition	RTW_OP_HMMUL	—	Yes, including the ability to map to BLAS multiplication functions

Target Function Libraries General Workflow

The general steps for creating and using a target function library are as follows:

- 1 Create one or more TFL tables containing replacement entries for math operators (+, −, *, /) and functions using an API based on the MATLAB API.

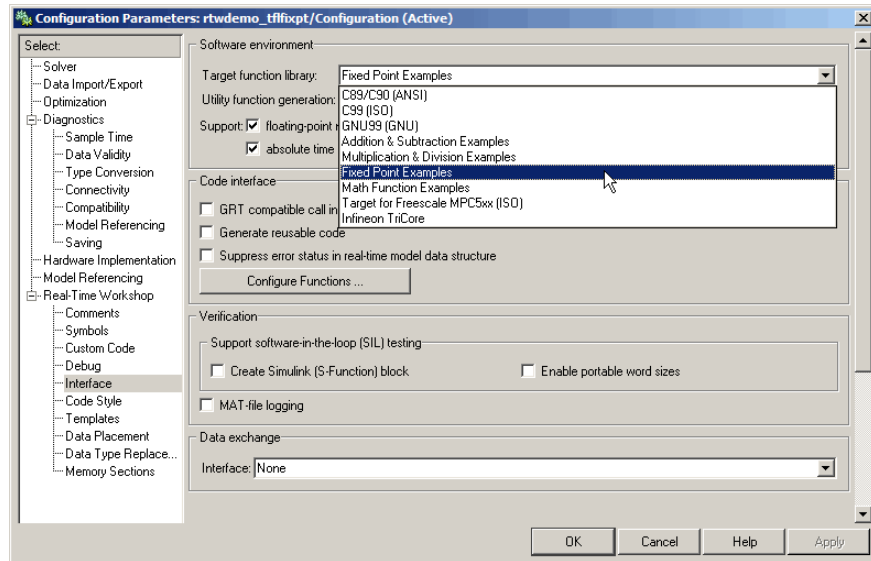
8. TFLs that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.

(The demo `rtwdemo_tf1_script` provides example tables that can be used as a starting point for customization.)

Name	Implementation	NumIn	In1Type	In2Type	RTW_OP_MUL																												
RTW_OP_ADD	u16_add_SameSlopeZeroBias	2	uint16	uint16	<table border="1"> <thead> <tr> <th colspan="2">RTW_OP_MUL</th> </tr> <tr> <th>General Information</th> <th>Fixed-point Settings</th> </tr> </thead> <tbody> <tr> <td colspan="2">Summary</td> </tr> <tr> <td>Description:</td> <td></td> </tr> <tr> <td>Key:</td> <td>RTW_OP_MUL with 2 input(s)</td> </tr> <tr> <td>Implementation:</td> <td>s16_mul_s16_s16_fac0125 () with 2 input(s)</td> </tr> <tr> <td>Implementation type:</td> <td>FCN_IMPL_FUNCT</td> </tr> <tr> <td>Saturation mode:</td> <td>RTW_WRAP_ON_OVERFLOW</td> </tr> <tr> <td>Rounding mode:</td> <td>RTW_ROUND_SIMPLEST</td> </tr> <tr> <td>GenCallback file:</td> <td></td> </tr> <tr> <td>Implementation header:</td> <td>s16_mul_s16_s16_fac0125.h</td> </tr> <tr> <td>Implementation source:</td> <td>s16_mul_s16_s16_fac0125.c</td> </tr> <tr> <td>Priority:</td> <td>90</td> </tr> <tr> <td>Total Usage Count:</td> <td>0</td> </tr> </tbody> </table>	RTW_OP_MUL		General Information	Fixed-point Settings	Summary		Description:		Key:	RTW_OP_MUL with 2 input(s)	Implementation:	s16_mul_s16_s16_fac0125 () with 2 input(s)	Implementation type:	FCN_IMPL_FUNCT	Saturation mode:	RTW_WRAP_ON_OVERFLOW	Rounding mode:	RTW_ROUND_SIMPLEST	GenCallback file:		Implementation header:	s16_mul_s16_s16_fac0125.h	Implementation source:	s16_mul_s16_s16_fac0125.c	Priority:	90	Total Usage Count:	0
RTW_OP_MUL																																	
General Information	Fixed-point Settings																																
Summary																																	
Description:																																	
Key:	RTW_OP_MUL with 2 input(s)																																
Implementation:	s16_mul_s16_s16_fac0125 () with 2 input(s)																																
Implementation type:	FCN_IMPL_FUNCT																																
Saturation mode:	RTW_WRAP_ON_OVERFLOW																																
Rounding mode:	RTW_ROUND_SIMPLEST																																
GenCallback file:																																	
Implementation header:	s16_mul_s16_s16_fac0125.h																																
Implementation source:	s16_mul_s16_s16_fac0125.c																																
Priority:	90																																
Total Usage Count:	0																																
RTW_OP_ADD	s32_add_SameSlopeZeroBias	2	int32	int32																													
RTW_OP_DIV	s16_div_s16_s16_slopebias	2	fixdt[1,16,15,2]	fixdt[1]																													
RTW_OP_DIV	s32_div_s16_s16_binarypoint	2	fixdt[1,16,15]	fixdt[1]																													
RTW_OP_DIV	s16_div_s16_s16_fac0125	2	int16	int16																													
RTW_OP_MINUS	s8_sub_SameSlopeZeroBias	2	int8	int8																													
RTW_OP_MUL	s32_mul_s16_s16_binarypoint	2	fixdt[1,16,15]	fixdt[1]																													
RTW_OP_MUL	s16_mul_s16_s16_fac0125	2	int16	int16																													

- 2 Register a target function library, consisting of one or more replacement tables, for use with Simulink or Embedded MATLAB Coder software. The MATLAB APIs `sl_customization` and `rtwTargetInfo` are provided for this purpose.
- 3 Open your Simulink model and select the desired target function library from the **Target function library** drop-down list located on the **Interface** pane of the Configuration Parameters dialog box. For Embedded MATLAB Coder applications, instantiate a Real-Time Workshop configuration object, set the Target Function Library, and provide the configuration object in a call to the `emlc` function, as follows:

```
rtwConfig = emlcoder.RTWConfig('ert');
rtwConfig.TargetFunctionLibrary = 'Addition & Subtraction Examples';
emlc -T rtw -s rtwConfig -c addsub_two_int16 -eg {t, t};
```



4 Build your Simulink model or Embedded MATLAB Coder application.

See the demo `rtwdemo_tf1_script`, which illustrates how to use TFLs to replace operators and functions in generated code. With each example model included in this demo, a separate TFL is provided to illustrate the creation of operator and function replacements and how to register the replacements with Simulink software.

Target Function Libraries Quick-Start Example

This section steps you through a simple example of the complete TFL workflow. (The materials for this example can easily be created based on the file and model displays in this section.)

- 1 Create and save a TFL table definition file that instantiates and populates a TFL table entry, such as the file `tf1_table_sinfcn.m` shown below. This file creates function table entries for the `sin` function. For detailed information on creating table definition files for math functions and operators, see “Creating Function Replacement Tables” on page 29-16.

```
function hTable = tf1_table_sinfcn()
%TFL_TABLE_SINFCN - Describe function entries for a Target Function Library table.
```

```

hTable = RTW.Tf1Table;

% Create entry for double data type sine function replacement
hTable.registerCFFunctionEntry(100, 1, 'sin', 'double', 'sin_dbl', ...
                               'double', '<sin_dbl.h>', '', '');

% Create entry for single data type sine function replacement
hTable.registerCFFunctionEntry(100, 1, 'sin', 'single', 'sin_sgl', ...
                               'double', '<sin_sgl.h>', '', '');

```

Note See “Example: Mapping Math Functions to Target-Specific Implementations” on page 29-27 for another example of `sin` function replacement, in which function arguments are created individually.

- 2 As a first check of the validity of your table entries, invoke the TFL table definition file as follows:

```

>> tbl = tf1_table_sinfcn

tbl =

RTW.Tf1Table
    Version: '1.0'
  AllEntries: [2x1 RTW.Tf1CFunctionEntry]
ReservedSymbols: []
StringResolutionMap: []
>>

```

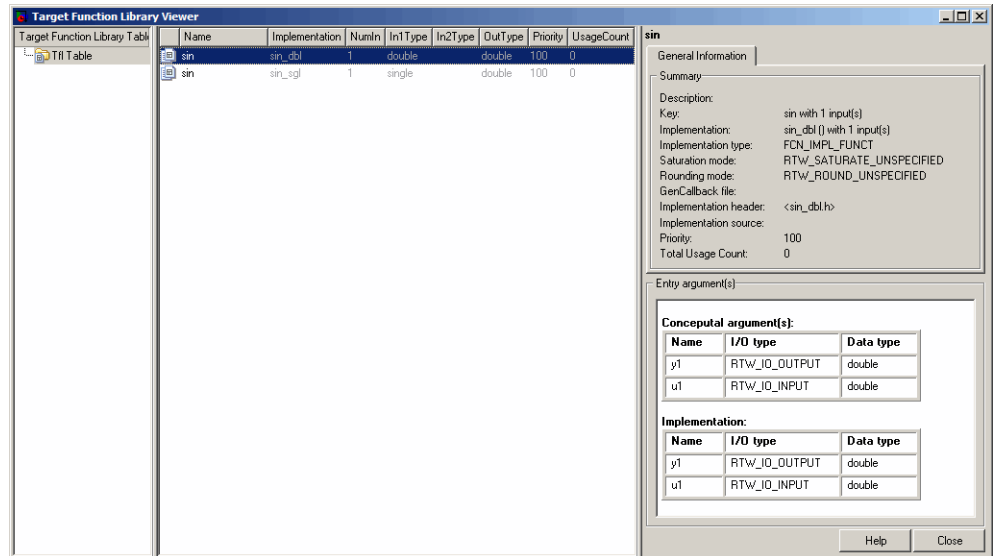
Any errors found during the invocation are displayed.

- 3 As a further check of your table entries, invoke the TFL Viewer using the following MATLAB command:

```

>> RTW.viewTf1(tf1_table_sinfcn)

```



Select entries in your table and verify that the graphical display of the contents of your table meets your expectations. (The TFL Viewer can also help you debug issues with the order of entries in a table, the order of tables in a TFL, and function signature mismatches. For more information, see “Examining and Validating Function Replacement Tables” on page 29-140.)

- 4 Create and save a TFL registration file that includes the `tf1_table_sinfcn` table, such as the `sl_customization.m` file shown below. The file specifies that the TFL to be registered is named 'Sine Function Example' and consists of `tf1_table_sinfcn`, with the default ANSI⁹ math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION
```

9. ANSI[®] is a registered trademark of the American National Standards Institute, Inc.

```
% Local function to define a TFL containing tfl_table_sinfcn
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties
thisTfl.Name = 'Sine Function Example';
thisTfl.Description = 'Demonstration of sine function replacement';
thisTfl.TableList = {'tfl_table_sinfcn'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

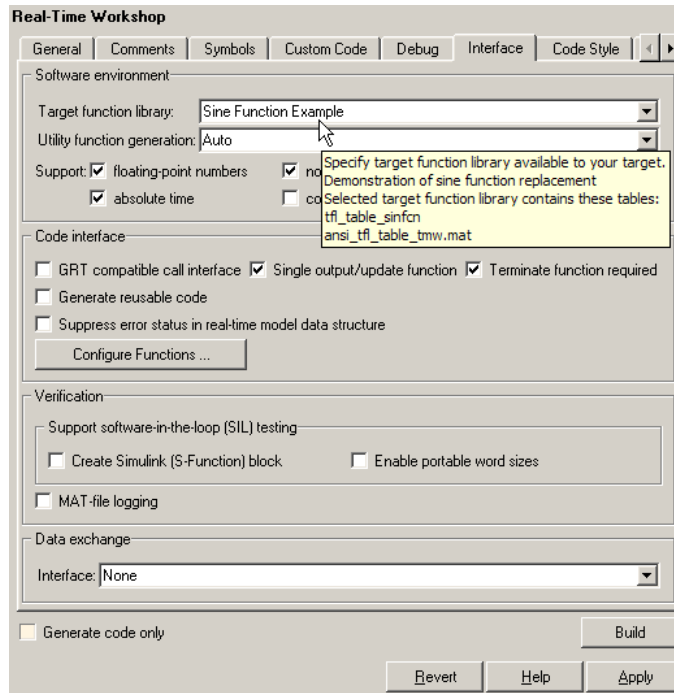
If you place this `s1_customization.m` file in the MATLAB search path or in the current working folder, the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `s1_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or Embedded MATLAB Coder software, see “Registering Target Function Libraries” on page 29-149.

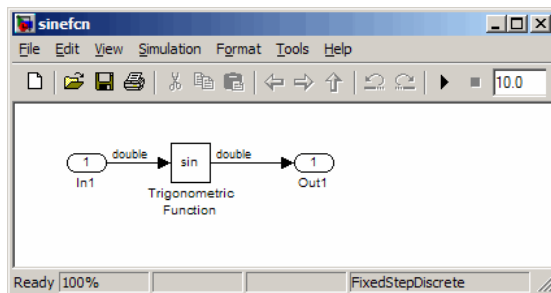
- 5 With your `s1_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip contains information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the command `RTW.viewTFL` with no argument, to examine all registered TFLs, including Sine Function Example.

- 6 Create an ERT-based model with a Trigonometric Function block set to the sine function, such as the following:



Make sure that the TFL you registered, Sine Function Example, is selected for this model.

- 7** Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Real-Time Workshop** pane, select the **Generate code only** option, and generate code for the model.
- 8** Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the Trigonometric Function block and select **Real-Time Workshop > Navigate to Code**. This selection highlights the `sin` function code within the model step function in `sinefcn.c`. In this case, `sin` has been replaced with `sin_dbl` in the generated code.

```

21
22 /* Real-time model */
23 RT_MODEL_sinefcn sinefcn_M;
24 RT_MODEL_sinefcn *sinefcn_M = &sinefcn_M;
25
26 /* Model step function */
27 void sinefcn_step(void)
28 {
29     /* Output: '<Root>/Out1' incorporates:
30      * Inport: '<Root>/In1'
31      * Trigonometry: '<Root>/Trigonometric Function'
32      */
33     sinefcn_Y.Out1 = sin_dbl(sinefcn_U.In1);
34 }
35
36 /* Model initialize function */
37 void sinefcn_initialize(void)
38 {
39     /* Registration code */

```

- 9 If functions were not replaced as you intended, you can use the techniques described in “Examining and Validating Function Replacement Tables” on page 29-140 to help you determine why the code generation process was unable to match a function signature with the TFL table entry you created for it.

For example, you can view the TFL cache hits and misses logged during the most recent build. For the code generation step in this example, there was one cache hit and zero cache misses, as shown in the following HitCache and MissCache entries:

```
>> a=get_param( sinefcn , TargetFcnLibHandle )

a =

RTW.TflControl
    Version: '1.0'
    HitCache: [1x1 RTW.TflFunctionEntry]
    MissCache: [0x1 handle]
    TLCCallList: [0x1 handle]
    TflTables: [2x1 RTW.TflTable]

>> a.HitCache(1)

ans =

RTW.TflFunctionEntry
    Key: 'sin'
    Priority: 100
    ConceptualArgs: [2x1 RTW.TflArgNumeric]
    Implementation: [1x1 RTW.CImplementation]
.
.
.
>>
```

Creating Function Replacement Tables

In this section...

“Overview of Function Replacement Table Creation” on page 29-16

“Creating Table Entries” on page 29-20

“Example: Mapping Math Functions to Target-Specific Implementations” on page 29-27

“Example: Mapping the memcpy Function to a Target-Specific Implementation” on page 29-34

“Example: Mapping Nonfinite Support Utility Functions to Target-Specific Implementations” on page 29-39

“Example: Mapping Scalar Operators to Target-Specific Implementations” on page 29-44

“Mapping Nonscalar Operators to Target-Specific Implementations” on page 29-50

“Mapping Fixed-Point Operators to Target-Specific Implementations” on page 29-80

“Remapping Operator Outputs to Implementation Function Input Positions” on page 29-115

“Refining TFL Matching and Replacement Using Custom TFL Table Entries” on page 29-117

“Specifying Build Information for Function Replacements” on page 29-134

“Adding Target Function Library Reserved Identifiers” on page 29-137

Overview of Function Replacement Table Creation

To create a TFL table containing replacement information for supported functions and operators, you perform the following steps:

- 1 Create a table definition file containing a function definition in the following general form:

```
function hTable = tfl_table_name()
    %TFL_TABLE_NAME - Describe entries for a Target Function Library table.
```

```
.
.
.
```

For example, the following sample function definition is from the “Target Function Libraries Quick-Start Example” on page 29-9:

```
function hTable = tfl_table_sinfcn()
%TFL_TABLE_SINFCN - Describe function entries for a Target Function Library table.
.
.
.
```

- 2** Within the function body, instantiate a TFL table with a command such as the following:

```
hTable = RTW.TflTable;
```

- 3** Use the TFL table creation functions (listed in the table below) to add table entries representing your replacements for supported functions and operators. For each individual function or operator entry, you issue one or more function calls to
 - a** Instantiate a table entry.
 - b** Add conceptual arguments, implementation arguments, and other attributes to the entry.
 - c** Add the entry to the table.

“Creating Table Entries” on page 29-20 describes this procedure in detail, including two methods for creating function entries. The following sample function entry is from the “Target Function Libraries Quick-Start Example” on page 29-9:

```
% Create entry for double data type sine function replacement
hTable.registerCFunctionEntry(100, 1, 'sin', 'double', 'sin_dbl', ...
                             'double', '<sin_dbl.h>', '', '');
```

- 4** Save the table definition file using the name of the table definition function, for example, `tfl_table_sinfcn.m`.

After you have created a table definition file, you can do the following:

- Examine and validate the table, as described in “Examining and Validating Function Replacement Tables” on page 29-140.
- Register a TFL containing the table with the Simulink software, as described in “Registering Target Function Libraries” on page 29-149.

After you register a TFL with the Simulink software, it appears in the Simulink GUI and can be selected for use in building models.

The following table provides a functional grouping of the TFL table creation functions.

Function	Description
Table entry creation	
addEntry	Add table entry to collection of table entries registered in TFL table
copyConceptualArgsToImplementation	Copy conceptual argument specifications to matching implementation arguments for TFL table entry
createAndAddConceptualArg	Create conceptual argument from specified properties and add to conceptual arguments for TFL table entry
createAndAddImplementationArg	Create implementation argument from specified properties and add to implementation arguments for TFL table entry
createAndSetCImplementationReturn	Create implementation return argument from specified properties and add to implementation for TFL table entry
enableCPP	Enable C++ support for function entry in TFL table
setNameSpace	Set name space for C++ function entry in TFL table
setTflCFunctionEntryParameters	Set specified parameters for function entry in TFL table
setTflCOperationEntryParameters	Set specified parameters for operator entry in TFL table
Alternative method for conceptual argument creation	

Function	Description
addConceptualArg	Add conceptual argument to array of conceptual arguments for TFL table entry
getTflArgFromString	Create TFL argument based on specified name and built-in data type
Alternative method for function entry creation	
registerCFunctionEntry	Create TFL function entry based on specified parameters and register in TFL table
registerCPPFunctionEntry	Create TFL C++ function entry based on specified parameters and register in TFL table
registerCPromotableMacroEntry	Create TFL promotable macro entry based on specified parameters and register in TFL table (for abs function replacement only)
Build information	
addAdditionalHeaderFile	Add additional header file to array of additional header files for TFL table entry
addAdditionalIncludePath	Add additional include path to array of additional include paths for TFL table entry
addAdditionalLinkObj	Add additional link object to array of additional link objects for TFL table entry
addAdditionalLinkObjPath	Add additional link object path to array of additional link object paths for TFL table entry
addAdditionalSourceFile	Add additional source file to array of additional source files for TFL table entry
addAdditionalSourcePath	Add additional source path to array of additional source paths for TFL table entry
Reserved identifiers	
setReservedIdentifiers	Register specified reserved identifiers to be associated with TFL table

Creating Table Entries

- “Overview of Table Entry Creation” on page 29-20
- “General Method for Creating Function and Operator Entries” on page 29-22
- “Alternative Method for Creating Function Entries” on page 29-26

Overview of Table Entry Creation

You define TFL table entries by issuing TFL table creation function calls inside a table definition file. The function calls must follow a function declaration and a TFL table instantiation, such as the following:

```
function hTable = tfl_table_sinfcn()
    %TFL_TABLE_SINFCON - Describe function entries for a Target Function Library table.

    hTable = RTW.TflTable;
```

Within the function body, you use the TFL table creation functions to add table entries representing your replacements for supported functions and operators. For each individual function or operator entry, you issue one or more function calls to

- 1 Instantiate a table entry.
- 2 Add conceptual arguments, implementation arguments, and other attributes to the entry.
- 3 Add the entry to the table.

The general method for creating function and operator entries, described in “General Method for Creating Function and Operator Entries” on page 29-22, uses the functions shown in the following table.

Function	Description
Table entry creation	
addEntry	Add table entry to collection of table entries registered in TFL table

Function	Description
copyConceptualArgsToImplementation	Copy conceptual argument specifications to matching implementation arguments for TFL table entry
createAndAddConceptualArg	Create conceptual argument from specified properties and add to conceptual arguments for TFL table entry
createAndAddImplementationArg	Create implementation argument from specified properties and add to implementation arguments for TFL table entry
createAndSetCImplementationReturn	Create implementation return argument from specified properties and add to implementation for TFL table entry
enableCPP	Enable C++ support for function entry in TFL table
setNameSpace	Set name space for C++ function entry in TFL table
setTflCFunctionEntryParameters	Set specified parameters for function entry in TFL table
setTflCOperationEntryParameters	Set specified parameters for operator entry in TFL table
Alternative method for conceptual argument creation	
addConceptualArg	Add conceptual argument to array of conceptual arguments for TFL table entry
getTflArgFromString	Create TFL argument based on specified name and built-in data type

A simpler alternative creation method is available for function entries, with the constraints that input types must be uniform and implementation arguments must use default Simulink naming. The alternative method uses the following functions and is described in “Alternative Method for Creating Function Entries” on page 29-26.

Function	Description
Alternative method for function entry creation	
registerCFunctionEntry	Create TFL function entry based on specified parameters and register in TFL table

Function	Description
registerCPPFunctionEntry	Create TFL C++ function entry based on specified parameters and register in TFL table
registerCPromotableMacroEntry	Create TFL promotable macro entry based on specified parameters and register in TFL table (for abs function replacement only)

General Method for Creating Function and Operator Entries

The general workflow for creating TFL table entries applies equally to function and operator replacements, and involves the following steps.

Note

- You can remap operator outputs to implementation function inputs for operator replacement entries (see “Remapping Operator Outputs to Implementation Function Input Positions” on page 29-115). However, for function replacement entries, implementation argument order must match the conceptual argument order. Remapping the argument order in a function implementation is not supported.
- For function entries, if your implementations additionally meet the requirements that all input arguments are of the same type and your implementation arguments use default Simulink naming (return argument *y1* and input arguments *u*n**), you can use a simpler alternative method for creating the entries, as described in “Alternative Method for Creating Function Entries” on page 29-26.

1 Within the function body of your table definition file, instantiate a TFL table entry for a function or operator, using one of the following lines of code:

<code>fcn_entry = RTW.Tf1CFunctionEntry;</code>	Supports function replacement
<code>op_entry = RTW.Tf1COperationEntry;</code>	Supports operator replacement
<code>op_entry = RTW.Tf1COperationEntry- Generator;</code>	Provides relative scaling factor (RSF) fixed-point parameters, described in “Mapping Fixed-Point Operators to Target-Specific Implementations” on page 29-80, that are not available in <code>RTW.Tf1COperationEntry</code>
<code>op_entry = RTW.Tf1COperationEntry- Generator_NetSlope;</code>	Provides net slope parameters, described in “Mapping Fixed-Point Operators to Target-Specific Implementations” on page 29-80, that are not available in <code>RTW.Tf1COperationEntry</code>
<code>op_entry = RTW.Tf1CBlasEntry- Generator;</code>	Supports replacement of nonscalar operators with MathWorks BLAS functions, described in “Mapping Nonscalar Operators to Target-Specific Implementations” on page 29-50
<code>op_entry = RTW.Tf1CBlasEntry- Generator;</code>	Supports replacement of nonscalar operators with ANSI/ISO C BLAS functions, described in “Mapping Nonscalar Operators to Target-Specific Implementations” on page 29-50
<code>op_entry = <i>MyCustomOperationEntry</i>; (where <i>MyCustomOperationEntry</i> is a class derived from <code>RTW.Tf1COperationEntry</code>)</code>	Supports operator replacement using custom TFL table entries, described in “Refining TFL Matching and Replacement Using Custom TFL Table Entries” on page 29-117

2 Set the table entry parameters, which are passed in parameter/value pairs to one of the following functions:

- `setTf1CFunctionEntryParameters`

- `setTf1COperationEntryParameters`

For example:

```
setTf1CFunctionEntryParameters(fcn_entry, ...
                               'Key',          'sin', ...
                               'Priority',      30, ...
                               'ImplementationName', 'mySin', ...
                               'ImplementationHeaderFile', 'basicMath.h',...
                               'ImplementationSourceFile', 'basicMath.c');
```

For detailed descriptions of the settable function and operator attributes, see the `setTf1CFunctionEntryParameters` and `setTf1COperationEntryParameters` reference pages in the Real-Time Workshop Embedded Coder documentation.

- 3 Create and add conceptual arguments to the function or operator entry. Output arguments must precede input arguments, and the function signature (including argument naming, order, and attributes) must fulfill the signature match sought by function or operator callers. Conceptual argument names follow the default Simulink naming convention:
 - For return argument, `y1`
 - For input argument names, `u1`, `u2`, ..., `un`

You can create and add conceptual arguments in either of two ways:

- Call the `createAndAddConceptualArg` function to create the argument and add it to the table entry. For example:

```
createAndAddConceptualArg(fcn_entry, 'RTW.Tf1ArgNumeric', ...
                          'Name',      'y1',...
                          'IOType',    'RTW_IO_OUTPUT',...
                          'DataTypeMode', 'double');
```

- Call the `getTf1ArgFromString` function to create an argument based on a built-in data type, and then call the `addConceptualArg` function to add the argument to the table entry.

Note If you use `getTf1ArgFromString`, the `IOType` property of the created argument defaults to `'RTW_IO_INPUT'`, indicating an input argument. For an output argument, you must change the `IOType` value to `'RTW_IO_OUTPUT'` by directly assigning the argument property, as shown in the following example.

```
arg = getTf1ArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
```

4 Create and add implementation arguments, representing the signature of your implementation function, to the function or operator entry. The implementation argument order must match the conceptual argument order. You can create and add implementation arguments in either of two ways:

- Call the `copyConceptualArgsToImplementation` function to populate all of the implementation arguments as copies of the previously created conceptual arguments. For example:

```
copyConceptualArgsToImplementation(fcn_entry);
```

- Call the `createAndSetCImplementationReturn` function to create the implementation return argument and add it to the table entry, and then call the `createAndAddImplementationArg` function to individually create and add each of your implementation arguments. This method allows you to vary argument attributes, including argument naming, as long as conceptual argument order is maintained. For example:

```
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
                                   'Name',      'y1', ...
                                   'IOType',    'RTW_IO_OUTPUT', ...
                                   'IsSigned',   true, ...
                                   'WordLength', 32, ...
                                   'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
                               'Name',    'u1', ...
                               'IOType',  'RTW_IO_INPUT',...
```

```

        'IsSigned', true,...
        'WordLength', 32, ...
        'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
        'Name',      'u2', ...
        'IOType',    'RTW_IO_INPUT',...
        'IsSigned', true,...
        'WordLength', 32, ...
        'FractionLength', 0 );

```

- 5 Add the function or operator entry to the TFL table using the `addEntry` function. For example:

```
addEntry(hTable, fcn_entry);
```

For complete examples of function entries and operator entries created using the general method, see “Example: Mapping Math Functions to Target-Specific Implementations” on page 29-27 and “Example: Mapping Scalar Operators to Target-Specific Implementations” on page 29-44. For syntax examples, see the examples in the TFL table creation function reference pages in the Real-Time Workshop Embedded Coder documentation.

Alternative Method for Creating Function Entries

You can use a simpler alternative method for creating TFL function entries if your function implementation meets the following criteria:

- The implementation argument order matches the conceptual argument order.
- All input arguments are of the same type.
- The return argument name and all input argument names follow the default Simulink naming convention:
 - For the return argument, y_1
 - For input argument names, u_1, u_2, \dots, u_n

The alternative method for creating function entries involves a single step. Call one of the following functions to create and add conceptual and implementation arguments and register the function entry:

- registerCFunctionEntry
- registerCPPFunctionEntry
- registerCPromotableMacroEntry (use only for the abs function)

For example:

```
hTable = RTW.Tf1Table;

registerCFunctionEntry(hTable, 100, 1, 'sqrt', 'double', ...
    'sqrt', 'double', '<math.h>', '', '');
```

For detailed descriptions of the function arguments, see the registerCFunctionEntry, registerCPPFunctionEntry, and registerCPromotableMacroEntry reference pages in the Real-Time Workshop Embedded Coder documentation.

Example: Mapping Math Functions to Target-Specific Implementations

The Real-Time Workshop Embedded Coder software supports the following math functions for replacement with custom library functions using target function library (TFL) tables.

Math Function	Simulink Support	Stateflow Support	Embedded MATLAB and Embedded MATLAB Coder Support
abs	<ul style="list-style-type: none"> • Floating-point • Integer 	<ul style="list-style-type: none"> • Floating-point • Integer 	Floating-point
acos	Floating-point	Floating-point	Floating-point
acosh	Floating-point	Not available (NA)	Not replaceable (NR)
asin	Floating-point	Floating-point	Floating-point
asinh	Floating-point	NA	NR
atan	Floating-point	Floating-point	Floating-point
atan2	Floating-point	Floating-point	Floating-point

Math Function	Simulink Support	Stateflow Support	Embedded MATLAB and Embedded MATLAB Coder Support
atanh	Floating-point	NA	NR
ceil	Floating-point	Floating-point	Floating-point
cos	Floating-point	Floating-point	Floating-point
cosh	Floating-point	Floating-point	Floating-point
exactrSqrt	<ul style="list-style-type: none"> • Floating-point • Integer 	NA	NA
exp	Floating-point	Floating-point	Floating-point
fix	Floating-point	NA	NR
floor	Floating-point	Floating-point	Floating-point
hypot	Floating-point	NA	NR
ldexp	Floating-point	Floating-point	Floating-point
ln	Floating-point	NA	NA
log	Floating-point	Floating-point	Floating-point
log10	Floating-point	Floating-point	Floating-point
max	<ul style="list-style-type: none"> • Floating-point • Integer 	<ul style="list-style-type: none"> • Floating-point • Integer 	<ul style="list-style-type: none"> • Floating-point • Integer
min	<ul style="list-style-type: none"> • Floating-point • Integer 	<ul style="list-style-type: none"> • Floating-point • Integer 	<ul style="list-style-type: none"> • Floating-point • Integer
mod/fmod	<ul style="list-style-type: none"> • Floating-point (mod) • Integer (mod) 	Floating-point (fmod)	NR
pow	Floating-point	Floating-point	Floating-point
rem	Floating-point	NA	Floating-point
round	Floating-point	NA	NR
rSqrt	<ul style="list-style-type: none"> • Floating-point • Integer 	NA	NA

Math Function	Simulink Support	Stateflow Support	Embedded MATLAB and Embedded MATLAB Coder Support
saturate	<ul style="list-style-type: none"> Floating-point Integer 	NA	NA
sign	<ul style="list-style-type: none"> Floating-point Integer 	NA	NR
sin	Floating-point	Floating-point	Floating-point
sinh	Floating-point	Floating-point	Floating-point
sqrt	Floating-point	Floating-point	Floating-point
tan	Floating-point	Floating-point	Floating-point
tanh	Floating-point	Floating-point	Floating-point

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry for the `sin` function.

Note See “Target Function Libraries Quick-Start Example” on page 29-9 for another example of `sin` function replacement, in which function arguments are created using the simpler method described in “Alternative Method for Creating Function Entries” on page 29-26.

- 1 Create and save the following TFL table definition file, `tf1_table_sinfcn2.m`. This file defines a TFL table containing a function replacement entry for the `sin` function.

The function body sets selected sine function entry parameters, creates the `y1` and `u1` conceptual arguments individually, and then copies the conceptual arguments to the implementation arguments. Finally the function entry is added to the table.

```
function hTable = tf1_table_sinfcn2()
%TFL_TABLE_SINFCN2 - Describe function entry for a Target Function Library table.
```

```

hTable = RTW.Tf1Table;

% Create entry for sine function replacement
fcn_entry = RTW.Tf1CFunctionEntry;
setTf1CFunctionEntryParameters(fcn_entry, ...
                               'Key',          'sin', ...
                               'Priority',      30, ...
                               'ImplementationName', 'mySin', ...
                               'ImplementationHeaderFile', 'basicMath.h',...
                               'ImplementationSourceFile', 'basicMath.c');

createAndAddConceptualArg(fcn_entry, 'RTW.Tf1ArgNumeric', ...
                          'Name',      'y1',...
                          'IOType',    'RTW_IO_OUTPUT',...
                          'DataTypeMode', 'double');

createAndAddConceptualArg(fcn_entry, 'RTW.Tf1ArgNumeric', ...
                          'Name',      'u1', ...
                          'IOType',    'RTW_IO_INPUT',...
                          'DataTypeMode', 'double');

copyConceptualArgsToImplementation(fcn_entry);

addEntry(hTable, fcn_entry);

```

- 2** Optionally, perform a quick check of the validity of the function entry by invoking the table definition file at the MATLAB command line (`>> tbl = tf1_table_sinfcn2`) and by viewing it in the TFL Viewer (`>> RTW.viewTf1(tf1_table_sinfcn2)`). For more information about validating TFL tables, see “Examining and Validating Function Replacement Tables” on page 29-140.
- 3** Create and save the following TFL registration file, which references the `tf1_table_sinfcn2` table.

The file specifies that the TFL to be registered is named 'Sine Function Example 2' and consists of `tf1_table_sinfcn2`, with the default ANSI¹⁰ math library as the base TFL table.

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

```

function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION

% Local function to define a TFL containing tfl_table_sinfcn2
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties
thisTfl.Name = 'Sine Function Example 2';
thisTfl.Description = 'Demonstration of sine function replacement';
thisTfl.TableList = {'tfl_table_sinfcn2'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN

```

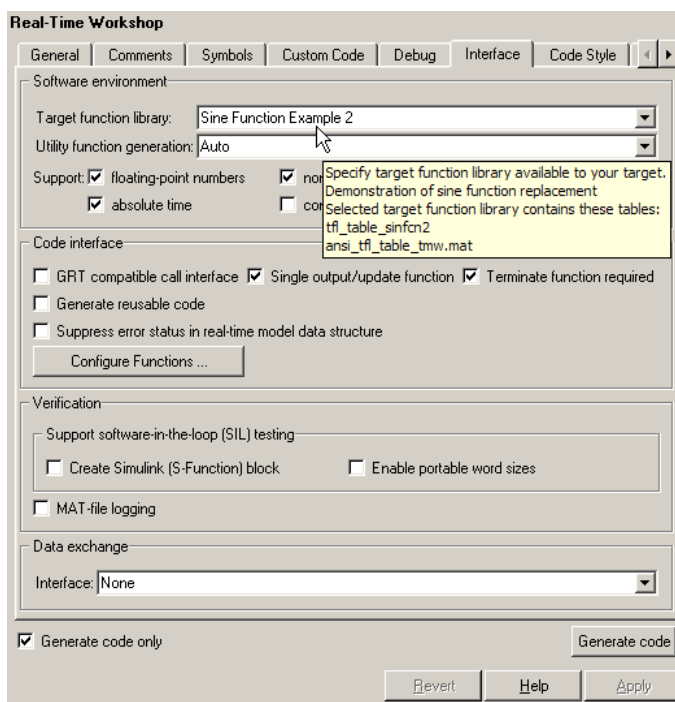
Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or Embedded MATLAB Coder software, see “Registering Target Function Libraries” on page 29-149.

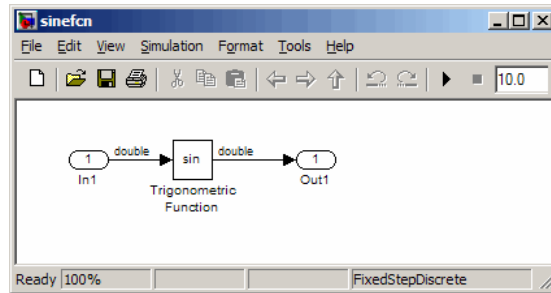
- 4 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including Sine Function Example 2.

- 5 Create an ERT-based model with a Trigonometric Function block set to the sine function, such as the following:



Make sure that the TFL you registered, Sine Function Example 2, is selected for this model.

- 6 Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Real-Time Workshop** pane, select the **Generate code only** option, and generate code for the model.
- 7 Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the Trigonometric Function block and select **Real-Time Workshop > Navigate to Code**. This selection highlights the `sin` function code within the model step function in `sinefcn.c`. In this case, `sin` has been replaced with `mySin` in the generated code.

```

21
22 /* Real-time model */
23 RT_MODEL_sinefcn sinefcn_M;
24 RT_MODEL_sinefcn *sinefcn_M = &sinefcn_M;
25
26 /* Model step function */
27 void sinefcn_step(void)
28 {
29     /* Outport: '<Root>/Out1' incorporates:
30      * Inport: '<Root>/In1'
31      * Trigonometry: '<Root>/Trigonometric Function'
32      */
33     sinefcn_Y.Out1 = mySin(sinefcn_U.In1);
34 }
35
36 /* Model initialize function */
37 void sinefcn_initialize(void)
38 {
39     /* Registration code */

```

Example: Mapping the memcpy Function to a Target-Specific Implementation

The Real-Time Workshop Embedded Coder software supports the memcpy function for replacement with custom library functions using target function library (TFL) tables.

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry for the memcpy function.

- 1 Create and save the following TFL table definition file, `tfl_table_memcpy.m`. This file defines a TFL table containing a function replacement entry for the memcpy function.

The function body sets selected memcpy function entry parameters, creates the y1, u1, u2, and u3 conceptual arguments individually, adds each argument to the conceptual arguments array for the function, and then copies the conceptual arguments to the implementation arguments. Finally the function entry is added to the table.

```

function hTable = tfl_table_memcpy()
%TFL_TABLE_MEMCPY - Describe memcpy function entry for a TFL table.

hTable = RTW.TflTable;

```

```

% Create function replacement entry for void* memcpy(void*, void*, size_t)
fcn_entry = RTW.Tf1CFunctionEntry;
setTf1CFunctionEntryParameters(fcn_entry, ...
                               'Key',          'memcpy', ...
                               'Priority',     90, ...
                               'ImplementationName', 'memcpy_int', ...
                               'ImplementationHeaderFile', 'memcpy_int.h',...
                               'SideEffects',  true);

% Set SideEffects to 'true' for function returning void to prevent it being
% optimized away

arg = getTf1ArgFromString(hTable, 'y1', 'void*');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(fcn_entry, arg);

arg = getTf1ArgFromString(hTable, 'u1', 'void*');
addConceptualArg(fcn_entry, arg);

arg = getTf1ArgFromString(hTable, 'u2', 'void*');
addConceptualArg(fcn_entry, arg);

arg = getTf1ArgFromString(hTable, 'u3', 'size_t');
addConceptualArg(fcn_entry, arg);

copyConceptualArgsToImplementation(fcn_entry);
addEntry(hTable, fcn_entry);

```

- 2** Optionally, perform a quick check of the validity of the `memcpy` entry by invoking the table definition file at the MATLAB command line (`>> tbl = tf1_table_memcpy`) and by viewing it in the TFL Viewer (`>> RTW.viewTf1(tf1_table_memcpy)`). For more information about validating TFL tables, see “Examining and Validating Function Replacement Tables” on page 29-140.
- 3** Create and save the following TFL registration file, which references the `tf1_table_memcpy` table.

The file specifies that the TFL to be registered is named 'Memcpy Function Example' and consists of `tfl_table_memcpy`, with the default ANSI¹¹ math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION

% Local function to define a TFL containing tfl_table_memcpy
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties
thisTfl.Name = 'Memcpy Function Example';
thisTfl.Description = 'Demonstration of memcpy function replacement';
thisTfl.TableList = {'tfl_table_memcpy'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

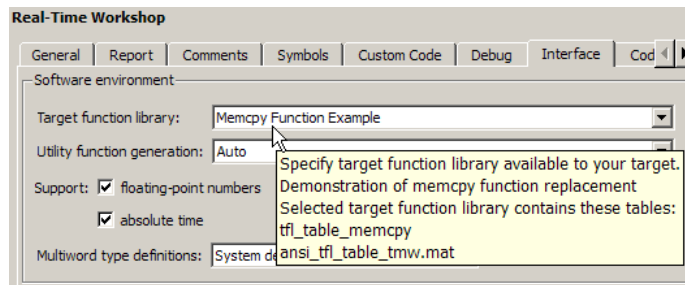
Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

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

For more information about registering TFLs with Simulink or Embedded MATLAB Coder software, see “Registering Target Function Libraries” on page 29-149.

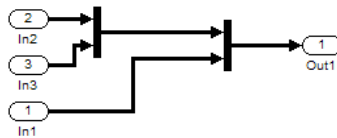
- 4 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.

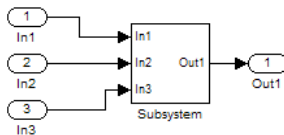


Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including Memcpy Function Example.

- 5 Create an ERT-based model that uses `memcpy` for vector assignments. For example,
 - a Use In, Out, and Mux blocks to create the following model. (Alternatively, you can open `rtwdemo_tflmath/Subsystem1` and copy the subsystem contents to a new model.)



- b Select the diagram and use **Edit > Subsystem** to make it a subsystem.



- c Select an ERT-based system target file on the **Real-Time Workshop** pane of the Configuration Parameters dialog box, and select the TFL you registered, **Memcpy Function Example**, on the **Interface** pane. You should also select a fixed-step solver on the **Solver** pane. Leave the memcpy options on the **Optimization** pane at their default settings, that is, **Use memcpy for vector assignment** selected, and **Memcpy threshold (bytes)** at 64. Apply the changes.
- d 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. Save the model. In this example, the model is saved to the name `memcpyfcn.mdl`.
- 6 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.
- 7 In the HTML code generation report, click on the `model.c` section (for example, `memcpyfcn.c`) and inspect the model step function to confirm that memcpy has been replaced with `memcpy_int` in the generated code.

```

31 /* Model step function */
32 void memcpyfcn_step(void)
33 {
34     /* Output: '<Root>/Out1' incorporates:
35      * Inport: '<Root>/In1'
36      * Inport: '<Root>/In2'
37      * Inport: '<Root>/In3'
38      */
39     memcpy_int((void *)(&memcpyfcn_Y.Out1[0]), (void *)(&memcpyfcn_U.In2[0]), 100U
40               * sizeof(int32_T));
41     memcpy_int((void *)(&memcpyfcn_Y.Out1[100]), (void *)(&memcpyfcn_U.In3[0]),
42               100U * sizeof(int32_T));
43     memcpy_int((void *)(&memcpyfcn_Y.Out1[200]), (void *)(&memcpyfcn_U.In1[0]),
44               100U * sizeof(int32_T));
45 }
46
47 /* Model initialize function */
48 void memcpyfcn_initialize(void)
49 {

```

Example: Mapping Nonfinite Support Utility Functions to Target-Specific Implementations

The Real-Time Workshop Embedded Coder software supports the following nonfinite support utility functions for replacement with custom library functions using target function library (TFL) tables.

GetInf
 GetMinusInf
 GetNaN

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create TFL table entries for the nonfinite functions.

- 1 Create and save the following TFL table definition file, `tfl_table_nonfinite.m`. This file defines a TFL table containing function replacement entries for the nonfinite functions.

For each nonfinite function, the function body uses the local function `locAddFcnEnt` to create entries for `single` and `double` replacement. For each entry, the local function sets selected function entry parameters, creates the `y1` and `u1` conceptual arguments individually, and then copies the conceptual arguments to the implementation arguments. Finally the function entry is added to the table.

```

function hTable = tfl_table_nonfinite()
%TFL_TABLE_NONFINITE - Describe function entries for a TFL table.

```

```

hTable = RTW.Tf1Table;

%% Create entries for nonfinite support utility functions
%locAddFcnEnt(hTable, key, implName, out, in1, hdr )
locAddFcnEnt(hTable, 'getNaN', 'getNaN', 'double', 'void', 'nonfin.h');
locAddFcnEnt(hTable, 'getNaN', 'getNaNF', 'single', 'void', 'nonfin.h');
locAddFcnEnt(hTable, 'getInf', 'getInf', 'double', 'void', 'nonfin.h');
locAddFcnEnt(hTable, 'getInf', 'getInfF', 'single', 'void', 'nonfin.h');
locAddFcnEnt(hTable, 'getMinusInf', 'getMinusInf', 'double', 'void', 'nonfin.h');
locAddFcnEnt(hTable, 'getMinusInf', 'getMinusInfF', 'single', 'void', 'nonfin.h');

%% Local Function
function locAddFcnEnt(hTable, key, implName, out, in1, hdr)
    if isempty(hTable)
        return;
    end

    fcn_entry = RTW.Tf1CFunctionEntry;
    setTf1CFunctionEntryParameters(fcn_entry, ...
        'Key', key, ...
        'Priority', 90, ...
        'ImplementationName', implName, ...
        'ImplementationHeaderFile', hdr);

    arg = getTf1ArgFromString(hTable, 'y1', out);
    arg.IOType = 'RTW_IO_OUTPUT';
    addConceptualArg(fcn_entry, arg);

    arg = getTf1ArgFromString(hTable, 'u1', in1);
    addConceptualArg(fcn_entry, arg);

    copyConceptualArgsToImplementation(fcn_entry);

    addEntry(hTable, fcn_entry);

%EOF

```

- 2 Optionally, perform a quick check of the validity of the nonfinite function entries by invoking the table definition file at the MATLAB command

line (`>> tbl = tfl_table_nonfinite`) and by viewing it in the TFL Viewer (`>> RTW.viewTfl(tfl_table_nonfinite)`). For more information about validating TFL tables, see “Examining and Validating Function Replacement Tables” on page 29-140.

- 3** Create and save the following TFL registration file, which references the `tfl_table_nonfinite` table.

The file specifies that the TFL to be registered is named 'Nonfinite Functions Example' and consists of `tfl_table_nonfinite`, with the default ANSI¹² math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION

% Local function to define a TFL containing tfl_table_nonfinite
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties
thisTfl.Name = 'Nonfinite Functions Example';
thisTfl.Description = 'Demonstration of nonfinite functions replacement';
thisTfl.TableList = {'tfl_table_nonfinite'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

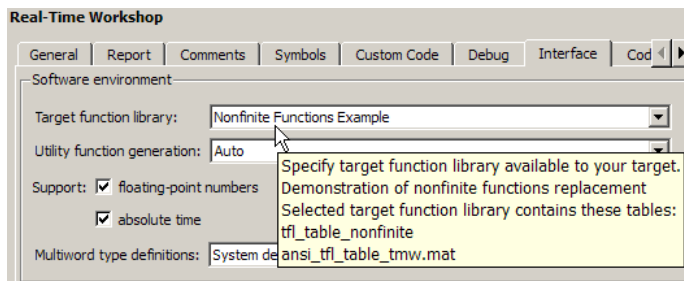
12. ANSI® is a registered trademark of the American National Standards Institute, Inc.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`;))

For more information about registering TFLs with Simulink or Embedded MATLAB Coder software, see “Registering Target Function Libraries” on page 29-149.

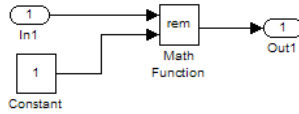
- 4 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Nonfinite Functions Example`.

- 5 Create an ERT-based model with a Math Function block set to the `rem` function, such as the following:



Open Model Explorer. Select the **Support: non-finite numbers** parameter on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box and configure the **Signal Attributes** for the In1 and Constant source blocks. For each source block, set **Data type** to **double**. Apply the changes. Save the model. In this example, the model is saved to the name `nonfinitefcns.mdl`.

Make sure that the TFL you registered, `Nonfinite Functions Example`, is selected for this model.

- 6** Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the option **Create code generation report**. Then go to the **Real-Time Workshop** pane, select the **Generate code only** option, and generate code for the model.
- 7** In the HTML code generation report, click on the `rtnonfinite.c` link and inspect the `rt_InitInfAndNaN` function to confirm that your replacements for nonfinite support functions are present in the generated code.

```

13  /*
14  * Abstract:
15  *   Real-Time Workshop function to initialize non-finites,
16  *   (Inf, NaN and -Inf).
17  */
18  #include "rt_ponfinite.h"
19  #include "nonfin.h"
20  #include "nonfin.h"
21  #define NumBitsPerChar      8
22
23  real_T rtInf;
24  real_T rtMinusInf;
25  real_T rtNaN;
26  real32_T rtInIF;
27  real32_T rtMinusInIF;
28  real32_T rtNaNF;
29
30  /* Function: rt_InitInfAndNaN =====
31  * Abstract:
32  *   Initialize the rtInf, rtMinusInf, and rtNaN needed by the
33  *   generated code. NaN is initialized as non-signaling. Assumes IEEE.
34  */
35  void rt_InitInfAndNaN(size_t realSize)
36  {
37      (void) (realSize);
38      rtNaN = getNaN();
39      rtNaNF = getNaNF();
40      rtInf = getInf();
41      rtInIF = getInIF();
42      rtMinusInf = getMinusInf();
43      rtMinusInIF = getMinusInIF();
44  }

```

Example: Mapping Scalar Operators to Target-Specific Implementations

The Real-Time Workshop Embedded Coder software supports the following scalar operators for replacement with custom library functions using target function library (TFL) tables:

Operator	Key
Addition (+)	RTW_OP_ADD
Subtraction (-)	RTW_OP_MINUS
Multiplication (*)	RTW_OP_MUL
Division (/)	RTW_OP_DIV
Data type conversion (cast)	RTW_OP_CAST
Shift left (<<) [integer and fixed-point data types]	RTW_OP_SL

Operator	Key
Shift right (>>) [integer and fixed-point data types]	RTW_OP_SRA (arithmetic) ¹³ RTW_OP_SRL (logical)
Complex conjugation	RTW_OP_CONJUGATE

Unless otherwise stated, the listed operators are supported for the following input data types:

- single, double, and their complex equivalents
- int8, int16, int32, and their complex equivalents
- uint8, uint16, uint32, and their complex equivalents
- Fixed-point data types
- Mixed data types (different types for different inputs)

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry for the + (addition) operator.

- 1 Create and save the following TFL table definition file, `tf1_table_add_uint8.m`. This file defines a TFL table containing an operator replacement entry for the + (addition) operator.

The function body sets selected addition operator entry parameters, creates the `y1`, `u1`, and `u2` conceptual arguments individually, and then copies the conceptual arguments to the implementation arguments. Finally, the operator entry is added to the table.

```
function hTable = tf1_table_add_uint8
%TFL_TABLE_ADD_UINT8 - Describe operator entry for a Target Function Library table.

hTable = RTW.Tf1Table;
```

13. TFLs that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.

```

% Create entry for addition of built-in uint8 data type
% Saturation on, Rounding no preference
op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingMode', 'RTW_ROUND_UNSPECIFIED', ...
    'ImplementationName', 'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = getTf1ArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);

arg = getTf1ArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(op_entry, arg );

arg = getTf1ArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(op_entry, arg );

copyConceptualArgsToImplementation(op_entry);

addEntry(hTable, op_entry);

```

- 2 Optionally, perform a quick check of the validity of the operator entry by invoking the table definition file at the MATLAB command line (>> `tbl = tf1_table_add_uint8`) and by viewing it in the TFL Viewer (>> `RTW.viewTfl(tf1_table_add_uint8)`).

For more information about validating TFL tables, see “Examining and Validating Function Replacement Tables” on page 29-140.

- 3 Create and save the following TFL registration file, which references the `tf1_table_add_uint8` table.

The file specifies that the TFL to be registered is named 'Addition Operator Example' and consists of `tf1_table_add_uint8`, with the default ANSI math library as the base TFL table.

```

function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION

% Local function to define a TFL containing tfl_table_add_uint8
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties
thisTfl.Name = 'Addition Operator Example';
thisTfl.Description = 'Demonstration of addition operator replacement';
thisTfl.TableList = {'tfl_table_add_uint8'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN

```

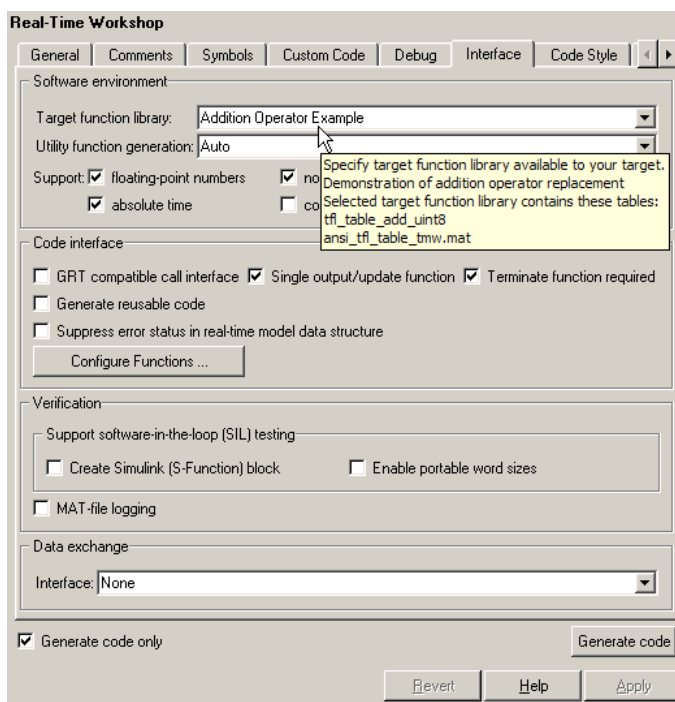
Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or Embedded MATLAB Coder software, see “Registering Target Function Libraries” on page 29-149.

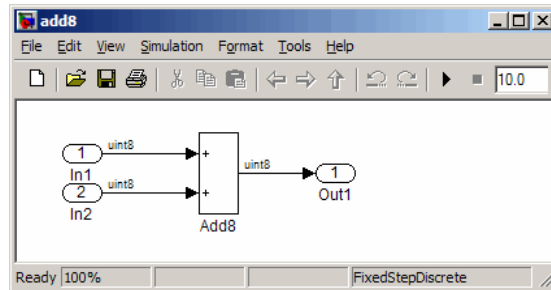
- 4 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including Addition Operator Example.

- 5 Create an ERT-based model with an Add block, such as the following:



Make sure that the TFL you registered, Addition Operator Example, is selected for this model.

- 6 Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Real-Time Workshop** pane, select the **Generate code only** option, and generate code for the model.
- 7 Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the Add block and select **Real-Time Workshop > Navigate to Code**. This selection highlights the Sum block code within the model step function in `add8.c`. In this case, code containing the `+` operator has been replaced with `u8_add_u8_u8` in the generated code.

```

21
22 /* Real-time model */
23 RT_MODEL_add8 add8_M;
24 RT_MODEL_add8 *add8_M = &add8_M;
25
26 /* Model step function */
27 void add8_step(void)
28 {
29     /* Output: '<Root>/Out1' incorporates:
30      * Inport: '<Root>/In1'
31      * Inport: '<Root>/In2'
32      * Sum: '<Root>/Add8'
33      */
34     add8_Y.Out1 = u8_add_u8_u8(add8_U.In1, add8_U.In2);
35 }
36
37 /* Model initialize function */
38 void add8_initialize(void)
39 {

```

Mapping Nonscalar Operators to Target-Specific Implementations

- “Example: Mapping Small Matrix Operations to Processor-Specific Intrinsic Functions” on page 29-51
- “Example: Mapping Matrix Multiplication to MathWorks BLAS Functions” on page 29-58
- “Example: Mapping Matrix Multiplication to ANSI/ISO C BLAS Functions” on page 29-69

The Real-Time Workshop Embedded Coder software supports the following nonscalar operators for replacement with custom library functions using target function library (TFL) tables:

Operator	Key
Addition (+)	RTW_OP_ADD
Subtraction (-)	RTW_OP_MINUS
Multiplication (*, .*)	RTW_OP_MUL
Complex conjugation	RTW_OP_CONJUGATE
Transposition (.')	RTW_OP_TRANS
Hermitian (complex conjugate) transposition (')	RTW_OP_HERMITIAN
Multiplication with transposition	RTW_OP_TRMUL
Multiplication with Hermitian transposition	RTW_OP_HMMUL

These operators are supported for the following input data types:

- single, double, and their complex equivalents
- int8, int16, int32, and their complex equivalents
- uint8, uint16, uint32, and their complex equivalents
- Fixed-point data types

- Mixed data types (different types for different inputs)

Note Saturation and rounding mode are ignored for floating-point nonscalar addition and subtraction. In TFL table entries for nonscalar addition and subtraction, if the argument data types are all floating-point, you should register 'RTW_ROUND_UNSPECIFIED' for the RoundingMode and SaturationMode parameters to setTflCOperationEntryParameters.

Example: Mapping Small Matrix Operations to Processor-Specific Intrinsic Functions

You can efficiently implement small matrix operations by invoking processor-specific intrinsic functions. The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry mapping small matrix sum operations to implementation functions that could invoke processor-specific intrinsic functions.

Note For examples of replacing other matrix operations and handling other data types, see the Matrix Operator Replacement section of the TFL demos page `rtwdemo_tfl_script`, including the demo model `rtwdemo_tflmatops` and its associated files.

- 1 Create and save the following TFL table definition file, `tfl_table_matrix_add_double.m`. This file defines a TFL table containing two matrix operator replacement entries for the + (addition) operator and the double data type.

The function body sets selected addition operator entry parameters, creates the `y1`, `u1`, and `u2` conceptual arguments individually, and then configures the implementation arguments. Finally, the operator entry is added to the table.

To specify a matrix argument to `createAndAddConceptualArg`, use the TFL argument class `RTW.TflArgMatrix` and specify the base type and the

dimensions for which the argument is valid. In this example, the first table entry specifies [2 2] and the second table entry specifies [3 3].

```
function hTable = tf1_table_matrix_add_double
%TFL_TABLE_MATRIX_ADD_DOUBLE - Describe two matrix operator entries for a TFL table.

hTable = RTW.Tf1Table;

LibPath = fullfile(matlabroot, 'toolbox', 'rtw', 'rtwdemos', 'tf1_demo');

% Create table entry for matrix_sum_2x2_double
op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 30, ...
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...
    'ImplementationName', 'matrix_sum_2x2_double', ...
    'ImplementationHeaderFile', 'MatrixMath.h', ...
    'ImplementationSourceFile', 'MatrixMath.c', ...
    'ImplementationHeaderPath', LibPath, ...
    'ImplementationSourcePath', LibPath, ...
    'AdditionalIncludePaths', {LibPath}, ...
    'GenCallback', 'RTW.copyFileToBuildDir', ...
    'SideEffects', true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix',...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix',...
    'Name', 'u2', ...
    'BaseType', 'double', ...
    'DimRange', [2 2]);
```



```

% Specify replacement function signature
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);
arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);
arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);
arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

% Create table entry for matrix_sum_3x3_double
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 30, ...
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...
    'ImplementationName', 'matrix_sum_3x3_double', ...
    'ImplementationHeaderFile', 'MatrixMath.h', ...
    'ImplementationSourceFile', 'MatrixMath.c', ...
    'ImplementationHeaderPath', LibPath, ...
    'ImplementationSourcePath', LibPath, ...
    'AdditionalIncludePaths', {LibPath}, ...
    'GenCallback', 'RTW.copyFileToBuildDir', ...
    'SideEffects', true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'BaseType', 'double', ...
    'DimRange', [3 3]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
    'Name', 'u1', ...
    'BaseType', 'double', ...
    'DimRange', [3 3]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...

```

```

        'Name',          'u2', ...
        'BaseType',    'double', ...
        'DimRange',    [3 3]);

% Specify replacement function signature
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);
arg = getTflArgFromString(hTable, 'u1', ['double' '*' ]);
op_entry.Implementation.addArgument(arg);
arg = getTflArgFromString(hTable, 'u2', ['double' '*' ]);
op_entry.Implementation.addArgument(arg);
arg = getTflArgFromString(hTable, 'y1', ['double' '*' ]);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

```

- 2** Optionally, perform a quick check of the validity of the operator entries by invoking the table definition file at the MATLAB command line (`>> tbl = tfl_table_matrix_add_double`) and by viewing it in the TFL Viewer (`>> RTW.viewTfl(tbl_table_matrix_add_double)`).

For more information about validating TFL tables, see “Examining and Validating Function Replacement Tables” on page 29-140.

- 3** Create and save the following TFL registration file, which references the `tfl_table_matrix_add_double` table.

The file specifies that the TFL to be registered is named 'Matrix Addition Operator Example' and consists of `tfl_table_matrix_add_double`, with the default ANSI math library as the base TFL table.

```

function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION

```

```

% Local function to define a TFL containing tfl_table_matrix_add_double
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties
thisTfl.Name = 'Matrix Addition Operator Example';
thisTfl.Description = 'Demonstration of matrix addition operator replacement';
thisTfl.TableList = {'tfl_table_matrix_add_double'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN

```

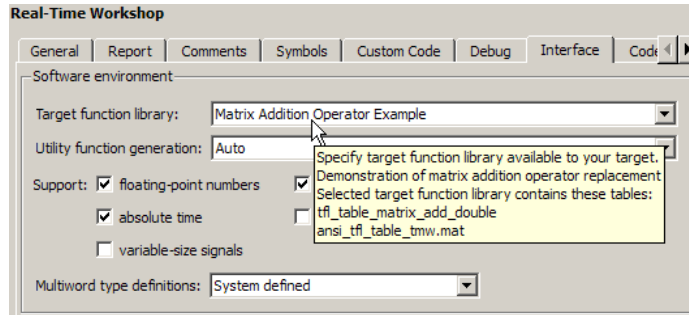
Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or Embedded MATLAB Coder software, see “Registering Target Function Libraries” on page 29-149.

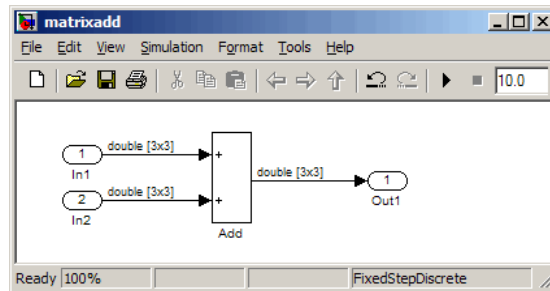
- 4 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including Matrix Addition Operator Example.

- 5 Create an ERT-based model with an Add block, such as the following:



Configure the **Signal Attributes** for the In1 and In2 source blocks. For each source block, set **Port dimensions** to [3 3] and set the **Data type** to double. Also, go to the **Solver** pane of the Configuration Parameters dialog box and select a fixed-step, discrete solver with a fixed-step size such as 0.1. Apply the changes. Save the model. In this example, the model is saved to the name `matrixadd.mdl`.

Make sure that the TFL you registered, Matrix Addition Operator Example, is selected for this model.

- 6 Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Real-Time Workshop** pane, select the **Generate code only** option, and generate code for the model.
- 7 Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the Add block and select **Real-Time Workshop > Navigate to Code**. This selection highlights the Sum block code within the model step function in matrixadd.c. In this case, code containing the + operator has been replaced with matrix_sum_3x3_double in the generated code.

```

26
27 /* Real-time model */
28 RT_MODEL_matrixadd matrixadd_M ;
29 RT_MODEL_matrixadd *matrixadd_M = &matrixadd_M ;
30
31 /* Model step function */
32 void matrixadd_step(void)
33 {
34     /* Output: '<Root>/Out1' incorporates:
35      * Inport: '<Root>/In1'
36      * Inport: '<Root>/In2'
37      * Sum: '<Root>/Add'
38     */
39     matrix_sum_3x3_double(&matrixadd_U.In1[0], &matrixadd_U.In2[0],
40                          &matrixadd_Y.Out1[0]);
41 }
42
43 /* Model initialize function */
44 void matrixadd_initialize(void)
45 {
46     /* Registration code */
47
48     /* initialize error status */
49     rtmSetErrorStatus(matrixadd_M, (NULL));
50

```

Note Optionally, you can reconfigure the In1 and In2 block **Port dimensions** to [2 2], regenerate code, and observe that code containing the + operator is replaced with matrix_sum_2x2_double.

Example: Mapping Matrix Multiplication to MathWorks BLAS Functions

You can use TFL tables to map nonscalar multiplication operations to the Basic Linear Algebra Subroutine (BLAS) multiplication functions `xgemm` and `xgemv`. The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry mapping floating-point matrix/matrix and matrix/vector multiplication operations to MathWorks BLAS library multiplication functions.

Note For examples of handling other data types, see the BLAS Support section of the TFL demos page `rtwdemo_tf1_script`, including the demo model `rtwdemo_tf1blas` and its associated files.

BLAS libraries support matrix/matrix multiplication in the form of $C = a(\text{op}(A) * \text{op}(B)) + bC$, where $\text{op}(X)$ means X , transposition of X , or Hermitian transposition of X . However, TFLs support only the limited case of $C = \text{op}(A) * \text{op}(B)$ ($a = 1.0$, $b = 0.0$). Correspondingly, although BLAS libraries support matrix/vector multiplication in the form of $y = a(\text{op}(A) * x) + by$, TFLs support only the limited case of $y = \text{op}(A) * x$ ($a = 1.0$, $b = 0.0$).

- 1 Create and save the following TFL table definition file, `tf1_table_tmwblas_mmult_double.m`. This file defines a TFL table containing `dgemm` and `dgemv` replacement entries for the matrix multiplication operator and the `double` data type.

For each entry, the function body sets selected matrix multiplication operator entry parameters, creates the `y1`, `u1`, and `u2` conceptual arguments individually, and then configures special implementation arguments that are required for `dgemm` and `dgemv` replacements. Finally, each operator entry is added to the table.

To specify a matrix argument to `createAndAddConceptualArg`, use the TFL argument class `RTW.Tf1ArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min`

... DimNMin; Dim1Max Dim2Max ... DimNMax]. For example, [2 2; inf inf] means any two-dimensional matrix of size 2x2 or larger. In this example, the conceptual output argument for the `dgemm32` entry for matrix/matrix multiplication replacement specifies dimensions [2 2; inf inf], while the conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions [2 1; inf 1].

```
function hTable = tfl_table_tmwblas_mmult_double
%TFL_TABLE_TMWBLAS_MMULT_DOUBLE - Describe two mmult operator entries for TFL table.

hTable = RTW.Tf1Table;

% Define library path for Windows or UNIX
arch = computer('arch');
if ~ispc
    LibPath = fullfile('$MATLAB_ROOT', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
        strcmp(compilerName, 'msvc80') || ...
        strcmp(compilerName, 'msvc71') || ...
        strcmp(compilerName, 'msvc60'), ...
        compilerName = 'microsoft';
    end
    LibPath = fullfile('$MATLAB_ROOT', 'extern', 'lib', arch, compilerName);
end

% Create table entry for dgemm32
op_entry = RTW.Tf1BlasEntryGenerator;
if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTf1COperationEntryParameters(op_entry, ...
```

```

        'Key',                    'RTW_OP_MUL', ...
        'Priority',              100, ...
        'ImplementationName',   'dgemm32', ...
        'ImplementationHeaderFile', 'blascompat32.h', ...
        'ImplementationHeaderPath', fullfile('${MATLAB_ROOT}','extern','include'), ...
        'AdditionalLinkObjs',   {'libmwblascompat32.' libExt}}, ...
        'AdditionalLinkObjsPaths', {LibPath}, ...
        'SideEffects',         true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'BaseType',      'double', ...
    'DimRange',      [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name',          'u1', ...
    'BaseType',      'double', ...
    'DimRange',      [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name',          'u2', ...
    'BaseType',      'double', ...
    'DimRange',      [1 1; inf inf]);

% Using RTW.Tf1BlasEntryGenerator for xgemm requires the following
% implementation signature:
%
% void f(char* TRANSA, char* TRANSB, int* M, int* N, int* K,
%       type* ALPHA, type* u1, int* LDA, type* u2, int* LDB,
%       type* BETA, type* y, int* LDC)
%
% Upon a successful match, the TFL entry will compute the correct
% values for M, N, K, LDA, LDB, and LDC and insert them into the
% generated code. TRANSA and TRANSB both will be set to 'N'.

% Specify replacement function signature

arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

```



```
arg = RTW.Tf1ArgCharConstant('TRANSA');
% Possible values for PassByType property are
% RTW_PASSBY_AUTO, RTW_PASSBY_POINTER,
% RTW_PASSBY_VOID_POINTER, RTW_PASSBY_BASE_POINTER
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = RTW.Tf1ArgCharConstant('TRANSB');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'K', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u1', ['double' '*'']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
```

```
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*' ]);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDB', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*' ]);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDC', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

% Create table entry for dgemv32
op_entry = RTW.TflBlasEntryGenerator;
if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTflCOperationEntryParameters(op_entry, ...
    'Key',                                'RTW_OP_MUL', ...
```

```

        'Priority',          100, ...
        'ImplementationName', 'dgemv32', ...
        'ImplementationHeaderFile', 'blascompat32.h', ...
        'ImplementationHeaderPath', fullfile('$MATLAB_ROOT','extern','include'), ...
        'AdditionalLinkObjs',    {'libmwblascompat32.' libExt}}, ...
        'AdditionalLinkObjsPaths', {LibPath},...
        'SideEffects',          true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'BaseType',      'double', ...
    'DimRange',      [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name',          'u1', ...
    'BaseType',      'double', ...
    'DimRange',      [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix',...
    'Name',          'u2', ...
    'BaseType',      'double', ...
    'DimRange',      [1 1; inf 1]);

% Using RTW.Tf1BlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(char* TRANS, int* M, int* N,
%         type* ALPHA, type* u1, int* LDA, type* u2, int* INCX,
%         type* BETA, type* y, int* INCY)
%
% Upon a successful match, the TFL entry will compute the correct
% values for M, N, LDA, INCX, and INCY, and insert them into the
% generated code. TRANS will be set to 'N'.

% Specify replacement function signature

arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

```

```
arg = RTW.Tf1ArgCharConstant('TRANS');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'INCX', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'BETA', 'double', 0);
```

```

arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*'']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCY', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

```

- 2** Optionally, perform a quick check of the validity of the operator entries by invoking the table definition file at the MATLAB command line (`>> tbl = tfl_table_tmwblas_mmult_double`) and by viewing it in the TFL Viewer (`>> RTW.viewTfl(tfl_table_tmwblas_mmult_double)`).

For more information about validating TFL tables, see “Examining and Validating Function Replacement Tables” on page 29-140.

- 3** Create and save the following TFL registration file, which references the `tfl_table_tmwblas_mmult_double` table.

The file specifies that the TFL to be registered is named 'MathWorks BLAS Matrix Multiplication Operator Example' and consists of `tfl_table_tmwblas_mmult_double`, with the default ANSI math library as the base TFL table.

```

function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION

```

```
% Local function to define a TFL containing tfl_table_tmwblas_mmult_double
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties
thisTfl.Name = 'MathWorks BLAS Matrix Multiplication Operator Example';
thisTfl.Description = 'Demonstration of MathWorks BLAS mmult operator replacement';
thisTfl.TableList = {'tfl_table_tmwblas_mmult_double'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

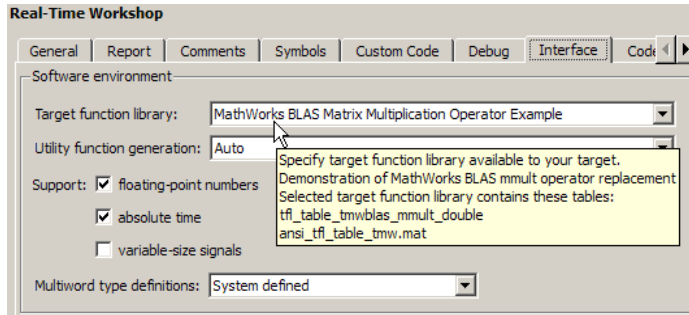
Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or Embedded MATLAB Coder software, see “Registering Target Function Libraries” on page 29-149.

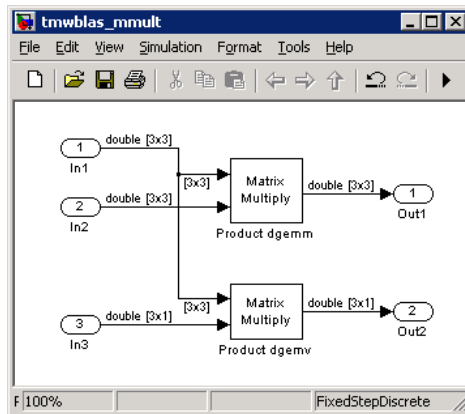
- 4 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including MathWorks BLAS Matrix Multiplication Operator Example.

5 Create an ERT-based model with two Product blocks, such as the following:



- a For each Product block, set the block parameter **Multiplication** to the value `Matrix(*)`.

- b** Configure the **Signal Attributes** for the In1, In2, and In3 source blocks. For In1 and In2, set **Port dimensions** to [3 3] and set the **Data type** to double. For In3, set **Port dimensions** to [3 1] and set the **Data type** to double.
 - c** Also, go to the **Solver** pane of the Configuration Parameters dialog box and select a fixed-step, discrete solver with a fixed-step size such as 0.1. Apply the changes.
 - d** Save the model. In this example, the model is saved to the name `tmwblas_mmult.mdl`.
 - e** Make sure that the TFL you registered, MathWorks BLAS Matrix Multiplication Operator Example, is selected for this model.
- 6** Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Real-Time Workshop** pane, select the **Generate code only** option, and generate code for the model.
- 7** Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the top Product block and select **Real-Time Workshop > Navigate to Code**. This selection highlights the Product block code within the model step function in `tmwblas_mmult.c`. In this case, code containing the matrix multiplication operator has been replaced with a call to `dgemm32` in the generated code.


```

45  /* Outport: '<Root>/Out1' incorporates:
46  * Inport: '<Root>/In1'
47  * Inport: '<Root>/In2'
48  * Product: '<Root>/Product dgemm'
49  */
50  TRANSA = 'N';
51  TRANSB = 'N';
52  M = 3;
53  N = 3;
54  K = 3;
55  ALPHA = 1.0;
56  LDA = 3;
57  LDB = 3;
58  BETA = 0.0;
59  LDC = 3;
60  dgemm32(&TRANSA, &TRANSB, &M, &N, &K, &ALPHA, &stmwblas_mmult_U.In1[0], &LDA,
61          &stmwblas_mmult_U.In2[0], &LDB, &BETA, &stmwblas_mmult_Y.Out1[0], &LDC);
62
63  /* Outport: '<Root>/Out2' incorporates:
64  * Inport: '<Root>/In1'
65  * Inport: '<Root>/In3'
66  * Product: '<Root>/Product dgemv'

```

Example: Mapping Matrix Multiplication to ANSI/ISO C BLAS Functions

You can use TFL tables to map nonscalar multiplication operations to the ANSI/ISO C BLAS multiplication functions `xgemm` and `xgemv`. The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry mapping floating-point matrix/matrix and matrix/vector multiplication operations to ANSI/ISO C BLAS library multiplication functions.

Note For examples of handling other data types, see the BLAS Support section of the TFL demos page `rtwdemo_tf1_script`, including the demo model `rtwdemo_tf1blas` and its associated files.

BLAS libraries support matrix/matrix multiplication in the form of

$C = a(\text{op}(A) * \text{op}(B)) + bC$, where $\text{op}(X)$ means X , transposition of X , or Hermitian transposition of X . However, TFLs support only the limited case of

$C = \text{op}(A) * \text{op}(B)$ ($a = 1.0$, $b = 0.0$). Correspondingly, although BLAS libraries

support matrix/vector multiplication in the form of $y = a(\text{op}(A) * x) + by$, TFLs support only the limited case of $y = \text{op}(A) * x$ ($a = 1.0$, $b = 0.0$).

- 1 Create and save the following TFL table definition file, `tfl_table_cblas_mmult_double.m`. This file defines a TFL table containing `dgemm` and `dgemv` replacement entries for the matrix multiplication operator and the `double` data type.

For each entry, the function body sets selected matrix multiplication operator entry parameters, creates the `y1`, `u1`, and `u2` conceptual arguments individually, and then configures special implementation arguments that are required for `dgemm` and `dgemv` replacements. Finally, each operator entry is added to the table.

To specify a matrix argument to `createAndAddConceptualArg`, use the TFL argument class `RTW.TflArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, `[2 2; inf inf]` means any two-dimensional matrix of size 2x2 or larger. In this example, the conceptual output argument for the `cblas_dgemm` entry for matrix/matrix multiplication replacement specifies dimensions `[2 2; inf inf]`, while the conceptual output argument for the `cblas_dgemv` entry for matrix/vector multiplication replacement specifies dimensions `[2 1; inf 1]`.

```
function hTable = tfl_table_cblas_mmult_double
%TFL_TABLE_CBLAS_MMULT_DOUBLE - Describe two mmult operator entries for TFL table.

hTable = RTW.TflTable;

LibPath = fullfile(matlabroot, 'toolbox', 'rtw', 'rtwdemos', 'tfl_demo');

% Create table entry for cblas_dgemm
op_entry = RTW.TflCblasEntryGenerator;
setTflCOperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'cblas_dgemm', ...
    'ImplementationHeaderFile', 'cblas.h', ...
```

```

    'ImplementationHeaderPath', LibPath, ...
    'AdditionalIncludePaths', {LibPath}, ...
    'GenCallback',            'RTW.copyFileToBuildDir', ...
    'SideEffects',            true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name',            'y1', ...
    'IOType',          'RTW_IO_OUTPUT', ...
    'BaseType',        'double', ...
    'DimRange',        [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name',            'u1', ...
    'BaseType',        'double', ...
    'DimRange',        [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgMatrix', ...
    'Name',            'u2', ...
    'BaseType',        'double', ...
    'DimRange',        [1 1; inf inf]);

% Using RTW.Tf1CBlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(enum ORDER, enum TRANSA, enum TRANSB, int M, int N, int K,
%         type ALPHA, type* u1, int LDA, type* u2, int LDB,
%         type BETA, type* y, int LDC)
%
% Since TFLs do not have the ability to specify enums, you must
% use integer. (This will cause problems with C++ code generation,
% so for C++, use a wrapper function to cast each int to the
% appropriate enumeration type.)
%
% Upon a successful match, the TFL entry will compute the correct
% values for M, N, K, LDA, LDB, and LDC and insert them into the
% generated code.

% Specify replacement function signature

arg = getTf1ArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';

```

```
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'ORDER', 'integer', 102);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'TRANSA', 'integer', 111);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'TRANSB', 'integer', 111);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'K', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 1);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDB', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
op_entry.Implementation.addArgument(arg);
```

```

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDC', 'integer', 0);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

% Create table entry for cblas_dgemv
op_entry = RTW.TflCBlasEntryGenerator;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                'RTW_OP_MUL', ...
    'Priority',           100, ...
    'ImplementationName', 'cblas_dgemv', ...
    'ImplementationHeaderFile', 'cblas.h', ...
    'ImplementationHeaderPath', LibPath, ...
    'AdditionalIncludePaths', {LibPath}, ...
    'GenCallback',        'RTW.copyFileToBuildDir', ...
    'SideEffects',        true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
    'Name',                'y1', ...
    'IOType',              'RTW_IO_OUTPUT', ...
    'BaseType',            'double', ...
    'DimRange',            [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
    'Name',                'u1', ...
    'BaseType',            'double', ...
    'DimRange',            [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
    'Name',                'u2', ...
    'BaseType',            'double', ...
    'DimRange',            [1 1; inf 1]);

% Using RTW.TflCBlasEntryGenerator for xgemv requires the following
% implementation signature:
%
```

```
% void f(enum ORDER, enum TRANSA, int M, int N,  
%       type ALPHA, type* u1, int LDA, type* u2, int INCX,  
%       type BETA, type* y, int INCY)  
%  
% Since TFLs do not have the ability to specify enums, you must  
% use integer. (This will cause problems with C++ code generation,  
% so for C++, use a wrapper function to cast each int to the  
% appropriate enumeration type.)  
%  
% Upon a successful match, the TFL entry will compute the correct  
% values for M, N, LDA, INCX, and INCY and insert them into the  
% generated code.  
  
% Specify replacement function signature  
  
arg = getTflArgFromString(hTable, 'y2', 'void');  
arg.IOType = 'RTW_IO_OUTPUT';  
op_entry.Implementation.setReturn(arg);  
  
arg = getTflArgFromString(hTable, 'ORDER', 'integer', 102);  
%arg.Type.ReadOnly=true;  
op_entry.Implementation.addArgument(arg);  
  
arg = getTflArgFromString(hTable, 'TRANSA', 'integer', 111);  
%arg.Type.ReadOnly=true;  
op_entry.Implementation.addArgument(arg);  
  
arg = getTflArgFromString(hTable, 'M', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTflArgFromString(hTable, 'N', 'integer', 0);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTflArgFromString(hTable, 'ALPHA', 'double', 1);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTflArgFromString(hTable, 'u1', ['double' '*'']);  
op_entry.Implementation.addArgument(arg);  
  
arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
```

```

op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCX', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCY', 'integer', 0);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

```

- 2** Optionally, perform a quick check of the validity of the operator entries by invoking the table definition file at the MATLAB command line (`>> tbl = tfl_table_cblas_mmult_double`) and by viewing it in the TFL Viewer (`>> RTW.viewTFL(tbl)`).

For more information about validating TFL tables, see “Examining and Validating Function Replacement Tables” on page 29-140.

- 3** Create and save the following TFL registration file, which references the `tfl_table_cblas_mmult_double` table.

The file specifies that the TFL to be registered is named 'ANSI/ISO C BLAS Matrix Multiplication Operator Example' and consists of `tfl_table_cblas_mmult_double`, with the default ANSI math library as the base TFL table.

```

function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

```

```
end % End of SL_CUSTOMIZATION

% Local function to define a TFL containing tfl_table_cblas_mmult_double
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties
thisTfl.Name = 'ANSI/ISO C BLAS Matrix Multiplication Operator Example';
thisTfl.Description = 'Demonstration of C BLAS mmult operator replacement';
thisTfl.TableList = {'tfl_table_cblas_mmult_double'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

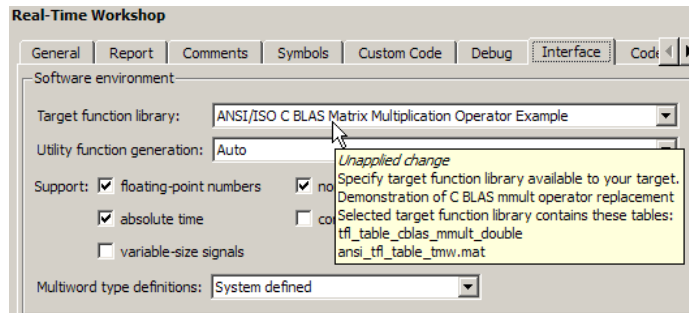
Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or Embedded MATLAB Coder software, see “Registering Target Function Libraries” on page 29-149.

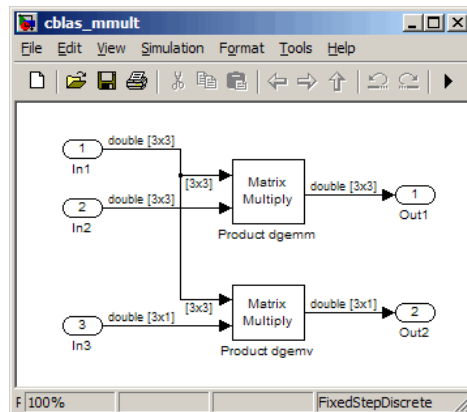
- 4 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including ANSI/ISO C BLAS Matrix Multiplication Operator Example.

5 Create an ERT-based model with two Product blocks, such as the following:



For each Product block, set the block parameter **Multiplication** to the value `Matrix(*)`.

- b** Configure the **Signal Attributes** for the In1, In2, and In3 source blocks. For In1 and In2, set **Port dimensions** to [3 3] and set the **Data type** to double. For In3, set **Port dimensions** to [3 1] and set the **Data type** to double.
 - c** Also, go to the **Solver** pane of the Configuration Parameters dialog box and select a fixed-step, discrete solver with a fixed-step size such as 0.1. Apply the changes.
 - d** Save the model. In this example, the model is saved to the name `cblas_mmult.mdl`.
 - e** Make sure that the TFL you registered, ANSI/ISO C BLAS Matrix Multiplication Operator Example, is selected for this model.
- 6** Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Real-Time Workshop** pane, select the **Generate code only** option, and generate code for the model.
- 7** Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the top Product block and select **Real-Time Workshop > Navigate to Code**. This selection highlights the Product block code within the model step function in `cblas_mmult.c`. In this case, code containing the matrix multiplication operator has been replaced with a call to `cblas_dgemm` in the generated code.

```
31 /* Model step function */
32 void cblas_mmult_step(void)
33 {
34     /* Outport: '<Root>/Out1' incorporates:
35      * Inport: '<Root>/In1'
36      * Inport: '<Root>/In2'
37      * Product: '<Root>/Product_dgemm'
38     */
39     cblas_dgemm(102, 111, 111, 3, 3, 3, 1.0, &cblas_mmult_U.In1[0], 3,
40                &cblas_mmult_U.In2[0], 3, 0.0, &cblas_mmult_Y.Out1[0], 3);
41
42     /* Outport: '<Root>/Out2' incorporates:
43      * Inport: '<Root>/In1'
44      * Inport: '<Root>/In3'
45      * Product: '<Root>/Product_dgemv'
46     */
47     cblas_dgemv(102, 111, 3, 3, 1.0, &cblas_mmult_U.In1[0], 3, &cblas_mmult_U.In3
48                [0], 1, 0.0, &cblas_mmult_Y.Out2[0], 1);
49 }
50
51 /* Model initialize function */
52 void cblas_mmult_initialize(void)
53 {
54     /* Registration code */
```

Mapping Fixed-Point Operators to Target-Specific Implementations

- “Overview of Fixed-Point Operator Replacement” on page 29-80
- “Fixed-Point Numbers and Arithmetic” on page 29-82
- “Creating Fixed-Point Operator Entries” on page 29-88
- “Example: Creating Fixed-Point Operator Entries for Binary-Point-Only Scaling” on page 29-91
- “Example: Creating Fixed-Point Operator Entries for [Slope Bias] Scaling” on page 29-94
- “Example: Creating Fixed-Point Operator Entries for Relative Scaling (Multiplication and Division)” on page 29-97
- “Example: Creating Fixed-Point Operator Entries for Net Slope (Multiplication and Division)” on page 29-100
- “Example: Creating Fixed-Point Operator Entries for Equal Slope and Zero Net Bias (Addition and Subtraction)” on page 29-104
- “Mapping Data Type Conversion (Cast) Operations to Target-Specific Implementations” on page 29-107
- “Mapping Fixed-Point Shift Left Operations to Target-Specific Implementations” on page 29-111

Overview of Fixed-Point Operator Replacement

The Real-Time Workshop Embedded Coder software supports TFL-based function replacement for the following scalar operations on fixed-point data types:

Operator	Key
Addition (+)	RTW_OP_ADD
Subtraction (-)	RTW_OP_MINUS
Multiplication (*)	RTW_OP_MUL
Division (/)	RTW_OP_DIV
Data type conversion (cast)	RTW_OP_CAST

Operator	Key
Shift left (<<)	RTW_OP_SL
Shift right (>>)	RTW_OP_SRA (arithmetic) ¹⁴ RTW_OP_SRL (logical)

Fixed-point operator table entries can be defined as matching:

- A specific binary-point-only scaling combination on the operator inputs and output.
- A specific [slope bias] scaling combination on the operator inputs and output.
- Relative scaling or net slope between multiplication or division operator inputs and output.

Use these methods to map a range of slope and bias values to a replacement function for multiplication or division.

- Equal slope and zero net bias across addition or subtraction operator inputs and output.

Use this method to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

14. TFLs that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.

Note

- The demo `rtwdemo_tflfixpt` demonstrates these replacements and provides example tables that can be used as a starting point for customization.
 - Using fixed-point data types in a model requires a Simulink Fixed Point license.
 - The fixed-point terminology used in this section is defined and explained in the *Simulink Fixed Point User's Guide*. See especially “Fixed-Point Numbers” and “Arithmetic Operations”.
-

Fixed-Point Numbers and Arithmetic

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V = \tilde{V} = SQ + B$$

where

- V is an arbitrarily precise real-world value.
- \tilde{V} is the approximate real-world value that results from fixed-point representation.
- Q is an integer that encodes \tilde{V} , referred to as the *quantized integer*.
- S is a coefficient of Q , referred to as the *slope*.
- B is an additive correction, referred to as the *bias*.

The general equation for an operation between fixed-point operands is as follows:

$$(S_o Q_o + B_o) \langle op \rangle (S_2 Q_2 + B_2)$$

The objective of TFL fixed-point operator replacement is to replace an operator that accepts and returns fixed-point or integer inputs and output with a function that accepts and returns built-in C numeric data types (not fixed-point data types). The following sections provide additional programming information for each supported operator.

Addition

The operation $V_0 = V_1 + V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1 + \left(\frac{S_2}{S_0} \right) Q_2 + \left(\frac{B_1 + B_2 - B_0}{S_0} \right)$$

If an addition replacement function is defined such that the scaling on the operands and sum are equal and the net bias

$$\left(\frac{B_1 + B_2 - B_0}{S_0} \right)$$

is zero (for example, a function `s8_add_s8_s8` that adds two signed 8-bit values and produces a signed 8-bit result), then the TFL operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to true. (For parameter descriptions, see the reference page for the function `setTflOperationEntryParameters`.)

Subtraction

The operation $V_0 = V_1 - V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1 - \left(\frac{S_2}{S_0} \right) Q_2 + \left(\frac{B_1 - B_2 - B_0}{S_0} \right)$$

If a subtraction replacement function is defined such that the scaling on the operands and difference are equal and the net bias

$$\left(\frac{B_1 - B_2 - B_0}{S_0} \right)$$

is zero (for example, a function `s8_sub_s8_s8` that subtracts two signed 8-bit values and produces a signed 8-bit result), then the TFL operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to true. (For parameter descriptions, see the reference page for the function `setTflOperationEntryParameters`.)

Multiplication

There are different ways to specify multiplication replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. For this, use the `TflOperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different TFL entry. For this, use a relative scaling factor (RSF) entry or a net slope entry:

- **Relative scaling factor (RSF) entry:**

The operation $V_0 = V_1 * V_2$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = (S_1 Q_1)(S_2 Q_2)$$

$$Q_0 = \left(\frac{S_1 S_2}{S_0} \right) Q_1 Q_2$$

$$Q_0 = S_n Q_1 Q_2$$

where S_n is the net slope.

Multiplication replacement functions may be defined such that all scaling is contained by a single operand. For example, a replacement function `s8_mul_s8_u8_rsf0p125` can multiply a signed 8-bit value by a factor of [0 ... 0.1245] and produce a signed 8-bit result. The following discussion describes how to convert the slope on each operand into a net factor.

To match a multiplication operation to the `s8_mul_s8_u8_rsf0p125` replacement function, $0 \leq S_n Q_2 \leq 2^{-3}$. Substituting the maximum integer value for Q_2 results in the following match criteria: When $S_n 2^8 = 2^{-3}$, or $S_n = 2^{-11}$, TFL replacement processing maps the multiplication operation to the `s8_mul_s8_u8_rsf0p125` function.

To accomplish this mapping, the TFL operator entry must define a *relative scaling factor*, $F2^E$, where the values for F and E are provided using operator entry parameters `RelativeScalingFactorF` and `RelativeScalingFactorE`. (For parameter descriptions, see the reference page for the function `setTf1COperationEntryParameters`.) For the `s8_mul_s8_u8_rsf0p125` function, the `RelativeScalingFactorF` would be set to 1 and the `RelativeScalingFactorE` would be set to -3.

Note When an operator entry specifies `RelativeScalingFactorF` and `RelativeScalingFactorE`, zero bias is implied for the inputs and output.

- **Net slope entry:**

Net slope entries are similar to the relative scaling factor entry described above. The difference is the match criteria. For a net slope entry, the net slope of the call-site operation, S_n , must match the specified net slope, $S_n = F2^E$, without regard to the maximum integer value. Specify the desired net slope F and E values using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTf1COperationEntryParameters`.)

Note When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Division

There are different ways to specify division replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. For this,

use the `Tf1COperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different TFL entry. For this, use a relative scaling factor (RSF) entry or a net slope entry:

- **Relative scaling factor (RSF) entry:**

The operation $V_0 = (V_1 / V_2)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{S_2 Q_2} \right)$$

$$Q_0 = S_n \left(\frac{Q_1}{Q_2} \right)$$

where S_n is the net slope.

As with multiplication, division replacement functions may be defined such that all scaling is contained by a single operand. For example, a replacement function `s16_rsf0p5_div_s16_s16` can divide a signed 16<<16 value by a signed 16-bit value and produce a signed 16-bit result. The following discussion describes how to convert the slope on each operand into a net factor.

To match a division operation to the `s16_rsf0p5_div_s16_s16` replacement function, $0 \leq S_n Q_1 \leq 2^{-1}$. Substituting the maximum integer value for Q_1 results in the following match criteria: When $S_n 2^{15} = 2^{-1}$, or $S_n = 2^{-16}$, TFL replacement processing maps the division operation to the `s8_mul_s8_u8_rsf0p125` function.

To accomplish this mapping, the TFL operator entry must define a *relative scaling factor*, $F2^E$, where the values for F and E are provided using operator entry parameters `RelativeScalingFactorF` and `RelativeScalingFactorE`. (For parameter descriptions, see the reference page for the function `setTf1COperationEntryParameters`.) For the `s16_rsf0p5_div_s16_s16` function, the `RelativeScalingFactorF` would be set to 1 and the `RelativeScalingFactorE` would be set to -1.

Note When an operator entry specifies `RelativeScalingFactorF` and `RelativeScalingFactorE`, zero bias is implied for the inputs and output.

- **Net slope entry:**

Net slope entries are similar to the relative scaling factor entry described above. The difference is the match criteria. For a net slope entry, the net slope of the call-site operation, S_n , must match the specified net slope, $S_n = F2^E$, without regard to the maximum integer value. Specify the desired net slope F and E values using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTf1COperationEntryParameters`.)

Note When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Data Type Conversion (Cast)

The data type conversion operation $V_0 = V_1$ implies, for binary-point-only scaling, that

$$Q_0 = \left(\frac{S_1}{S_0} \right) Q_1$$

$$Q_0 = S_n Q_1$$

where S_n is the net slope.

Shift

The shift left or shift right operation $V_0 = (V_1 / 2^n)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{2^n} \right)$$

$$Q_0 = \left(\frac{S_1}{S_0} \right) + \left(\frac{Q_1}{2^n} \right)$$

$$Q_0 = S_n \left(\frac{Q_1}{2^n} \right)$$

where S_n is the net slope.

Creating Fixed-Point Operator Entries

To create TFL table entries for fixed-point operators, you use the “General Method for Creating Function and Operator Entries” on page 29-22 and specify fixed-point parameter/value pairs to the functions shown in the following table.

Function	Description
<code>createAndAddConceptualArg</code>	Create conceptual argument from specified properties and add to conceptual arguments for TFL table entry
<code>createAndAddImplementationArg</code>	Create implementation argument from specified properties and add to implementation arguments for TFL table entry
<code>createAndSetCImplementationReturn</code>	Create implementation return argument from specified properties and add to implementation for TFL table entry
<code>setTf1COperationEntryParameters</code>	Set specified parameters for operator entry in TFL table

The following table maps some common methods of matching TFL fixed-point operator table entries to the associated fixed-point parameters that you need to specify in your TFL table definition file.

To match...	Instantiate class...	Minimally specify parameters...
<p>A specific binary-point-only scaling combination on the operator inputs and output</p> <p>See “Example: Creating Fixed-Point Operator Entries for Binary-Point-Only Scaling” on page 29-91.</p>	<p>RTW.Tf1COperationEntry</p>	<p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value true. • CheckBias: Specify the value true. • DataTypeMode (or DataType/Scaling equivalent): Specify fixed-point binary-point-only scaling. • FractionLength: Specify a fraction length (for example, 3).
<p>A specific [slope bias] scaling combination on the operator inputs and output</p> <p>See “Example: Creating Fixed-Point Operator Entries for [Slope Bias] Scaling” on page 29-94.</p>	<p>RTW.Tf1COperationEntry</p>	<p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value true. • CheckBias: Specify the value true. • DataTypeMode (or DataType/Scaling equivalent): Specify fixed-point [slope bias] scaling. • Slope (or SlopeAdjustmentFactor/-FixedExponent equivalent): Specify a slope value (for example, 15). • Bias: Specify a bias value (for example, 2).

To match...	Instantiate class...	Minimally specify parameters...
<p>Relative scaling between operator inputs and output (multiplication and division)</p> <p>See “Example: Creating Fixed-Point Operator Entries for Relative Scaling (Multiplication and Division)” on page 29-97.</p>	<p>RTW.Tf1COperationEntry-Generator</p>	<p>setTf1COperationEntryParameters function:</p> <ul style="list-style-type: none"> • RelativeScalingFactorF: Specify the slope adjustment factor (F) part of the relative scaling factor, $F2^E$ (for example, 1.0). • RelativeScalingFactorE: Specify the fixed exponent (E) part of the relative scaling factor, $F2^E$ (for example, -3.0). <p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value false. • CheckBias: Specify the value false. • DataType: Specify the value 'Fixed'.
<p>Net slope between operator inputs and output (multiplication and division)</p> <p>See “Example: Creating Fixed-Point Operator Entries for Net Slope (Multiplication and Division)” on page 29-100.</p>	<p>RTW.Tf1COperationEntry-Generator_NetSlope</p>	<p>setTf1COperationEntryParameters function:</p> <ul style="list-style-type: none"> • NetSlopeAdjustmentFactor: Specify the slope adjustment factor (F) part of the net slope, $F2^E$ (for example, 1.0). • NetFixedExponent: Specify the fixed exponent (E) part of the net slope, $F2^E$ (for example, -3.0). <p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value false. • CheckBias: Specify the value false. • DataType: Specify the value 'Fixed'.

To match...	Instantiate class...	Minimally specify parameters...
<p>Equal slope and zero net bias across operator inputs and output (addition and subtraction)</p> <p>See “Example: Creating Fixed-Point Operator Entries for Equal Slope and Zero Net Bias (Addition and Subtraction)” on page 29-104.</p>	<p>RTW.Tf1COperationEntry-Generator</p>	<p>setTf1COperationEntryParameters function:</p> <ul style="list-style-type: none"> • SlopesMustBeTheSame: Specify the value true. • MustHaveZeroNetBias: Specify the value true. <p>createAndAddConceptualArg function:</p> <ul style="list-style-type: none"> • CheckSlope: Specify the value false. • CheckBias: Specify the value false.

Example: Creating Fixed-Point Operator Entries for Binary-Point-Only Scaling

TFL table entries for operations on fixed-point data types can be defined as matching a specific binary-point-only scaling combination on the operator inputs and output. These binary-point-only scaling entries can map the specified binary-point-scaling combination to a replacement function for addition, subtraction, multiplication, or division.

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry for multiplication of fixed-point data types where arguments are specified with binary-point-only scaling. In this example:

- The TFL operator entry is instantiated using the RTW.Tf1COperationEntry class.
- The function setTf1COperationEntryParameters is called to set operator entry parameters. These parameters include the type of operation (multiplication), the saturation mode (saturate on overflow), the rounding mode (unspecified), and the name of the replacement function (s32_mul_s16_s16_binarypoint).

- The function `createAndAddConceptualArg` is called to create and add conceptual output and input arguments to the operator entry. Each argument specifies that the data type is fixed-point, the mode is binary-point-only scaling, and its derived slope and bias values must exactly match the call-site slope and bias values. The output argument is 32 bits, signed, with a fraction length of 28, while the input arguments are 16 bits, signed, with fraction lengths of 15 and 13.
- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output argument is 32 bits and signed (`int32`) and the input arguments are 16 bits and signed (`int16`).

```

hTable = RTW.Tf1Table;

op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key',                'RTW_OP_MUL', ...
    'Priority',            90, ...
    'SaturationMode',     'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingMode',       'RTW_ROUND_UNSPECIFIED', ...
    'ImplementationName', 's32_mul_s16_s16_binarypoint', ...
    'ImplementationHeaderFile', 's32_mul_s16_s16_binarypoint.h', ...
    'ImplementationSourceFile', 's32_mul_s16_s16_binarypoint.c');

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',                'y1', ...
    'IOType',              'RTW_IO_OUTPUT', ...
    'CheckSlope',          true, ...
    'CheckBias',           true, ...
    'DataTypeMode',        'Fixed-point: binary point scaling', ...
    'IsSigned',            true, ...
    'WordLength',          32, ...
    'FractionLength',      28);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',                'u1', ...
    'IOType',              'RTW_IO_INPUT', ...

```



```

        'CheckSlope',    true, ...
        'CheckBias',    true, ...
        'DataTypeMode', 'Fixed-point: binary point scaling', ...
        'IsSigned',     true, ...
        'WordLength',   16, ...
        'FractionLength', 15);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',            'u2', ...
    'IOType',          'RTW_IO_INPUT', ...
    'CheckSlope',     true, ...
    'CheckBias',      true, ...
    'DataTypeMode',   'Fixed-point: binary point scaling', ...
    'IsSigned',       true, ...
    'WordLength',     16, ...
    'FractionLength', 13);

createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',            'y1', ...
    'IOType',          'RTW_IO_OUTPUT', ...
    'IsSigned',       true, ...
    'WordLength',     32, ...
    'FractionLength', 0);

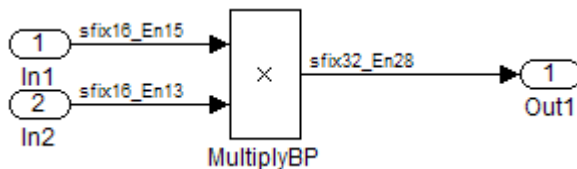
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',            'u1', ...
    'IOType',          'RTW_IO_INPUT', ...
    'IsSigned',       true, ...
    'WordLength',     16, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',            'u2', ...
    'IOType',          'RTW_IO_INPUT', ...
    'IsSigned',       true, ...
    'WordLength',     16, ...
    'FractionLength', 0);

addEntry(hTable, op_entry);

```

To generate code using this table entry, you can follow the general procedure in “Example: Mapping Scalar Operators to Target-Specific Implementations” on page 29-44, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to `fixdt(1,16,15)`
- Set the Inport 2 **Data type** to `fixdt(1,16,13)`
- In the Product block:
 - Set **Output data type** to `fixdt(1,32,28)`
 - Select the option **Saturate on integer overflow**

Example: Creating Fixed-Point Operator Entries for [Slope Bias] Scaling

TFL table entries for operations on fixed-point data types can be defined as matching a specific [slope bias] scaling combination on the operator inputs and output. These [slope bias] scaling entries can map the specified [slope bias] combination to a replacement function for addition, subtraction, multiplication, or division.

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry for division of fixed-point data types where arguments are specified using [slope bias] scaling. In this example:

- The TFL operator entry is instantiated using the `RTW.Tf1COperationEntry` class.
- The function `setTf1COperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation

(division), the saturation mode (saturate on overflow), the rounding mode (round to ceiling), and the name of the replacement function (`s16_div_s16_s16_slopebias`).

- The function `createAndAddConceptualArg` is called to create and add conceptual output and input arguments to the operator entry. Each argument specifies that the data type is fixed-point, the mode is [slope bias] scaling, and its specified slope and bias values must exactly match the call-site slope and bias values. The output argument and input arguments are 16 bits, signed, each with specific [slope bias] specifications.
- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```
hTable = RTW.Tf1Table;
```

```
op_entry = RTW.Tf1COperationEntry;
```

```
setTf1COperationEntryParameters(op_entry, ...
```

```
    'Key',                'RTW_OP_DIV', ...
    'Priority',           90, ...
    'SaturationMode',    'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingMode',      'RTW_ROUND_CEILING', ...
    'ImplementationName', 's16_div_s16_s16_slopebias', ...
    'ImplementationHeaderFile', 's16_div_s16_s16_slopebias.h', ...
    'ImplementationSourceFile', 's16_div_s16_s16_slopebias.c');
```

```
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
```

```
    'Name',                'y1', ...
    'IOType',              'RTW_IO_OUTPUT', ...
    'CheckSlope',          true, ...
    'CheckBias',           true, ...
    'DataTypeMode',        'Fixed-point: slope and bias scaling', ...
    'IsSigned',            true, ...
    'WordLength',          16, ...
    'Slope',                15, ...
    'Bias',                 2);
```

```

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'CheckSlope',   true, ...
    'CheckBias',    true, ...
    'DataTypeMode', 'Fixed-point: slope and bias scaling', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'Slope',        15, ...
    'Bias',         2);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u2', ...
    'IOType',       'RTW_IO_INPUT', ...
    'CheckSlope',   true, ...
    'CheckBias',    true, ...
    'DataTypeMode', 'Fixed-point: slope and bias scaling', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'Slope',        13, ...
    'Bias',         5);

createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',     true, ...
    'WordLength',   16, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'u2', ...
    'IOType',       'RTW_IO_INPUT', ...
    'IsSigned',     true, ...

```

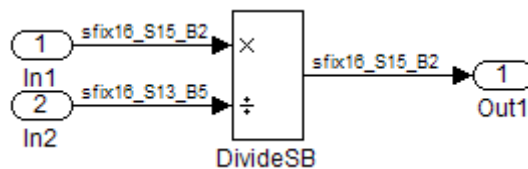
```

        'WordLength',      16, ...
        'FractionLength', 0);

    addEntry(hTable, op_entry);

```

To generate code using this table entry, you can follow the general procedure in “Example: Mapping Scalar Operators to Target-Specific Implementations” on page 29-44, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to `fixdt(1,16,15,2)`
- Set the Inport 2 **Data type** to `fixdt(1,16,13,5)`
- In the Divide block:
 - Set **Output data type** to **Inherit**: Inherit via back propagation
 - Set **Integer rounding mode** to **Ceiling**
 - Select the option **Saturate on integer overflow**

Example: Creating Fixed-Point Operator Entries for Relative Scaling (Multiplication and Division)

TFL table entries for multiplication or division of fixed-point data types can be defined as matching relative scaling between operator inputs and output. These relative scaling entries can map a range of slope and bias values to a replacement function for multiplication or division.

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry for division of fixed-point data types using a relative scaling factor. In this example:

- The TFL operator entry is instantiated using the `RTW.Tf1COperationEntryGenerator` class, which provides access to the fixed-point parameters `RelativeScalingFactorF` and `RelativeScalingFactorE`.
- The function `setTf1COperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (division), the saturation mode (saturation off), the rounding mode (round to ceiling), and the name of the replacement function (`s16_div_s16_s16_rsf0p125`). Additionally, `RelativeScalingFactorF` and `RelativeScalingFactorE` are used to specify the F and E parts of the relative scaling factor $F2^E$.
- The function `createAndAddConceptualArg` is called to create and add conceptual output and input arguments to the operator entry. Each argument is specified as fixed-point, 16 bits, and signed. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.
- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```

hTable = RTW.Tf1Table;

op_entry = RTW.Tf1COperationEntryGenerator;
setTf1COperationEntryParameters(op_entry, ...
                                'Key',          'RTW_OP_DIV', ...
                                'Priority',     90, ...
                                'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...
                                'RoundingMode', 'RTW_ROUND_CEILING', ...
                                'RelativeScalingFactorF', 1.0, ...
                                'RelativeScalingFactorE', -3.0, ...
                                'ImplementationName', 's16_div_s16_s16_rsf0p125', ...
                                'ImplementationHeaderFile', 's16_div_s16_s16_rsf0p125.h', ...
                                'ImplementationSourceFile', 's16_div_s16_s16_rsf0p125.c');

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...

```

```

        'Name',          'y1', ...
        'IOType',       'RTW_IO_OUTPUT', ...
        'CheckSlope',   false, ...
        'CheckBias',    false, ...
        'DataType',     'Fixed', ...
        'IsSigned',     true, ...
        'WordLength',   16);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
        'Name',          'u1', ...
        'IOType',       'RTW_IO_INPUT', ...
        'CheckSlope',   false, ...
        'CheckBias',    false, ...
        'DataType',     'Fixed', ...
        'IsSigned',     true, ...
        'WordLength',   16);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
        'Name',          'u2', ...
        'IOType',       'RTW_IO_INPUT', ...
        'CheckSlope',   false, ...
        'CheckBias',    false, ...
        'DataType',     'Fixed', ...
        'IsSigned',     true, ...
        'WordLength',   16);

createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
        'Name',          'y1', ...
        'IOType',       'RTW_IO_OUTPUT', ...
        'IsSigned',     true, ...
        'WordLength',   16, ...
        'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
        'Name',          'u1', ...
        'IOType',       'RTW_IO_INPUT', ...
        'IsSigned',     true, ...
        'WordLength',   16, ...
        'FractionLength', 0);

```

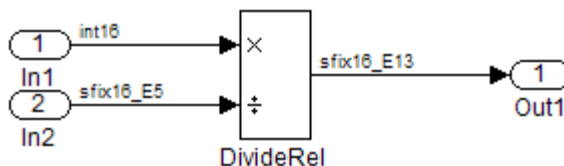
```

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
                                'Name',          'u2', ...
                                'IOType',        'RTW_IO_INPUT', ...
                                'IsSigned',      true, ...
                                'WordLength',    16, ...
                                'FractionLength', '0);

addEntry(hTable, op_entry);

```

To generate code using this table entry, you can follow the general procedure in “Example: Mapping Scalar Operators to Target-Specific Implementations” on page 29-44, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to int16
- Set the Inport 2 **Data type** to fixdt(1,16,-5)
- In the Divide block:
 - Set **Output data type** to fixdt(1,16,-13)
 - Set **Integer rounding mode** to Ceiling

Example: Creating Fixed-Point Operator Entries for Net Slope (Multiplication and Division)

TFL table entries for multiplication or division of fixed-point data types can be defined as matching net slope between operator inputs and output. These net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry for division of fixed-point data types using a net slope. In this example:

- The TFL operator entry is instantiated using the `RTW.Tf1COperationEntryGenerator_NetSlope` class, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.
- The function `setTf1COperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (division), the saturation mode (wrap on overflow), the rounding mode (unspecified), and the name of the replacement function (`user_div_*`). Additionally, `NetSlopeAdjustmentFactor` and `NetFixedExponent` are used to specify the F and E parts of the net slope $F2^E$.
- The function `createAndAddConceptualArg` is called to create and add conceptual output and input arguments to the operator entry. Each argument is specified as fixed-point and signed. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.
- The function `getTf1ArgFromString` is called to create implementation output and input arguments that are added to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```

hTable = RTW.Tf1Table;

wv = [16,32];
for iy = 1:2
    for inum = 1:2
        for iden = 1:2
            hTable = getDivOpEntry(hTable, ...
                                  fixdt(1,wv(iy)),fixdt(1,wv(inum)),fixdt(1,wv(iden)));
        end
    end
end

%-----

```

```

function hTable = getDivOpEntry(hTable,dty,dtnum,dtden)
%-----
% Create an entry for division of fixed-point data types where
% arguments are specified using Slope and Bias scaling
% Saturation on, Rounding unspecified

funcStr = sprintf('user_div_%s_%s_%s',...
    typeStrFunc(dty),...
    typeStrFunc(dtnum),...
    typeStrFunc(dtden));

op_entry = RTW.Tf1COperationEntryGenerator_NetSlope;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_DIV', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW',...
    'RoundingMode', 'RTW_ROUND_UNSPECIFIED',...
    'NetSlopeAdjustmentFactor', 1.0, ...
    'NetFixedExponent', 0.0, ...
    'ImplementationName', funcStr, ...
    'ImplementationHeaderFile', [funcStr,'.h'], ...
    'ImplementationSourceFile', [funcStr,'.c'] );

createAndAddConceptualArg(op_entry, ...
    'RTW.Tf1ArgNumeric', ...
    'Name', 'y1',...
    'IOType', 'RTW_IO_OUTPUT',...
    'CheckSlope', false,...
    'CheckBias', false,...
    'DataTypeMode', 'Fixed-point: slope and bias scaling',...
    'IsSigned', dty.Signed,...
    'WordLength', dty.WordLength,...
    'Bias', 0);

createAndAddConceptualArg(op_entry, ...
    'RTW.Tf1ArgNumeric',...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT',...
    'CheckSlope', false,...
    'CheckBias', false,...

```

```

        'DataTypeMode', 'Fixed-point: slope and bias scaling',...
        'IsSigned',     dtnum.Signed,...
        'WordLength',   dtnum.WordLength,...
        'Bias',         0);

createAndAddConceptualArg(op_entry, ...
    'RTW.Tf1ArgNumeric', ...
    'Name',             'u2', ...
    'IOType',          'RTW_IO_INPUT',...
    'CheckSlope',     false,...
    'CheckBias',      false,...
    'DataTypeMode',   'Fixed-point: slope and bias scaling',...
    'IsSigned',       dtden.Signed,...
    'WordLength',     dtden.WordLength,...
    'Bias',           0);

arg = getTf1ArgFromString(hTable, 'y1', typeStrBase(dty));
op_entry.Implementation.setReturn(arg);

arg = getTf1ArgFromString(hTable, 'u1', typeStrBase(dtnum));
op_entry.Implementation.addArgument(arg);

arg = getTf1ArgFromString(hTable, 'u2', typeStrBase(dtden));
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

%-----
function str = typeStrFunc(dt)
%-----

if dt.Signed
    sstr = 's';
else
    sstr = 'u';
end
str = sprintf('%s%d',sstr,dt.WordLength);

%-----
function str = typeStrBase(dt)

```

```

%-----

if dt.Signed
    sstr = ;
else
    sstr = 'u';
end
str = sprintf('%sint%d',sstr,dt.WordLength);

```

Example: Creating Fixed-Point Operator Entries for Equal Slope and Zero Net Bias (Addition and Subtraction)

TFL table entries for addition or subtraction of fixed-point data types can be defined as matching relative slope and bias values (equal slope and zero net bias) across operator inputs and output. These entries allow you to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry for addition of fixed-point data types where slopes must be equal and net bias must be zero across the operator inputs and output. In this example:

- The TFL operator entry is instantiated using the RTW.TflCOperationEntryGenerator class, which provides access to the fixed-point parameters SlopesMustBeTheSame and MustHaveZeroNetBias.
- The function setTflCOperationEntryParameters is called to set operator entry parameters. These parameters include the type of operation (addition), the saturation mode (saturation off), the rounding mode (unspecified), and the name of the replacement function (u16_add_SameSlopeZeroBias). Additionally, SlopesMustBeTheSame and MustHaveZeroNetBias are set to true to indicate that slopes must be equal and net bias must be zero across the addition inputs and output.
- The function createAndAddConceptualArg is called to create and add conceptual output and input arguments to the operator entry. Each argument is specified as 16 bits and unsigned. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.

- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and unsigned (`uint16`).

```

hTable = RTW.Tf1Table;

op_entry = RTW.Tf1COperationEntryGenerator;
setTf1COperationEntryParameters(op_entry, ...
    'Key',          'RTW_OP_ADD', ...
    'Priority',     90, ...
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingMode', 'RTW_ROUND_UNSPECIFIED', ...
    'SlopesMustBeTheSame', true, ...
    'MustHaveZeroNetBias', true, ...
    'ImplementationName', 'u16_add_SameSlopeZeroBias', ...
    'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
    'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'CheckSlope',    false, ...
    'CheckBias',     false, ...
    'IsSigned',      false, ...
    'WordLength',    16);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'CheckSlope',    false, ...
    'CheckBias',     false, ...
    'IsSigned',      false, ...
    'WordLength',    16);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u2', ...
    'IOType',        'RTW_IO_INPUT', ...

```

```

        'CheckSlope',    false, ...
        'CheckBias',    false, ...
        'IsSigned',     false, ...
        'WordLength',   16);

createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
        'Name',         'y1', ...
        'IOType',       'RTW_IO_OUTPUT', ...
        'IsSigned',     false, ...
        'WordLength',   16, ...
        'FractionLength', 0);

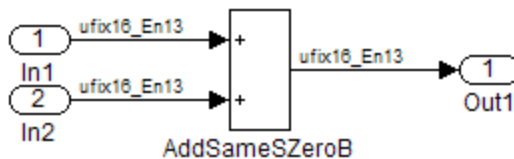
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
        'Name',         'u1', ...
        'IOType',       'RTW_IO_INPUT', ...
        'IsSigned',     false, ...
        'WordLength',   16, ...
        'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
        'Name',         'u2', ...
        'IOType',       'RTW_IO_INPUT', ...
        'IsSigned',     false, ...
        'WordLength',   16, ...
        'FractionLength', 0);

addEntry(hTable, op_entry);

```

To generate code using this table entry, you can follow the general procedure in “Example: Mapping Scalar Operators to Target-Specific Implementations” on page 29-44, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to `fixdt(0,16,13)`
- Set the Inport 2 **Data type** to `fixdt(0,16,13)`
- In the Add block:
 - Verify that **Output data type** is set to its default, `Inherit via internal rule`
 - Set **Integer rounding mode** to `Zero`

Mapping Data Type Conversion (Cast) Operations to Target-Specific Implementations

- “Example: Creating a TFL Entry to Replace Casts From `int32` To `int16`” on page 29-107
- “Example: Creating a TFL Entry to Replace Fixed-Point Casts Using `Net Slope`” on page 29-108

You can use TFL table entries to replace the default generated code for data type conversion (cast) operations with calls to optimized functions.

For details of the arithmetic supported for replacement of data type conversion, see the data type conversion (cast) subsection of “Fixed-Point Numbers and Arithmetic” on page 29-82.

Example: Creating a TFL Entry to Replace Casts From `int32` To `int16`.

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry to replace `int32` to `int16` data type conversion (cast) operations. In this example:

- The TFL operator entry is instantiated using the `RTW.Tf1COperationEntry` class.
- The function `setTf1COperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (cast), the saturation mode (saturate on overflow), the rounding mode (toward negative infinity), and the name of the replacement function (`my_sat_cast`).

- The function `getTflArgFromString` is called to create an `int16` output argument, which is then added to the operator entry both as the first conceptual argument and the implementation return argument.
- The function `getTflArgFromString` is called to create an `int32` input argument, which is then added to the operator entry both as the second conceptual argument and the sole implementation input argument.

```

hTable = RTW.TflTable;

% Create an int16 to int32 cast replacement
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                'RTW_OP_CAST', ...
    'Priority',           50, ...
    'ImplementationName', 'my_sat_cast', ...
    'SaturationMode',    'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingMode',     'RTW_ROUND_FLOOR', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

% Create int16 arg as conceptual arg 1 and implementation return
arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
op_entry.Implementation.setReturn(arg);

% Create int32 arg as conceptual arg 2 and implementation input arg 1
arg = getTflArgFromString(hTable, 'u1', 'int32');
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

```

Example: Creating a TFL Entry to Replace Fixed-Point Casts Using Net Slope. The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry to replace data type conversions (casts) of fixed-point data types using a net slope. In this example:

- The TFL operator entry is instantiated using the `RTW.Tf1COperationEntryGenerator_NetSlope` class, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.
- The function `setTf1COperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (cast), the saturation mode (saturate on overflow), the rounding mode (toward negative infinity), and the name of the replacement function (`my_fxp_cast`). Additionally, `NetSlopeAdjustmentFactor` and `NetFixedExponent` are used to specify the F and E parts of the net slope $F2^E$.
- The function `createAndAddConceptualArg` is called to create conceptual output and input arguments that are added to the operator entry. Each argument is specified as fixed-point and signed. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.
- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create implementation return and input arguments that are added to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```

hTable = RTW.Tf1Table;

% Create a fixed-point cast replacement using a NetSlope entry
op_entry = RTW.Tf1COperationEntryGenerator_NetSlope;
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
OutWL = 32;
OutSgn = true;
setTf1COperationEntryParameters(op_entry, ...
    'Key',                'RTW_OP_CAST', ...
    'Priority',            50, ...
    'SaturationMode',    'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingMode',      'RTW_ROUND_FLOOR', ...
    'NetSlopeAdjustmentFactor', 1.0, ...
    'NetFixedExponent',   (OutFL - InFL), ...

```

```

'SaturationMode',          'RTW_SATURATE_ON_OVERFLOW', ...
'RoundingMode',           'RTW_ROUND_FLOOR', ...
'ImplementationName',     'my_fxp_cast', ...
'ImplementationHeaderFile', 'some_hdr.h', ...
'ImplementationSourceFile', 'some_hdr.c');

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
                           'Name',          'y1', ...
                           'IOType',        'RTW_IO_OUTPUT', ...
                           'CheckSlope',    false, ...
                           'CheckBias',     false, ...
                           'DataTypeMode',  'Fixed-point: binary point scaling', ...
                           'IsSigned',      OutSgn, ...
                           'WordLength',    OutWL, ...
                           'FractionLength', OutFL);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
                           'Name',          'u1', ...
                           'IOType',        'RTW_IO_INPUT', ...
                           'CheckSlope',    false, ...
                           'CheckBias',     false, ...
                           'DataTypeMode',  'Fixed-point: binary point scaling', ...
                           'IsSigned',      InSgn, ...
                           'WordLength',    InWL, ...
                           'FractionLength', InFL);

createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
                                   'Name',          'y1', ...
                                   'IOType',        'RTW_IO_OUTPUT', ...
                                   'IsSigned',      OutSgn, ...
                                   'WordLength',    OutWL, ...
                                   'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
                               'Name',          'u1', ...
                               'IOType',        'RTW_IO_INPUT', ...
                               'IsSigned',      InSgn, ...
                               'WordLength',    InWL, ...
                               'FractionLength', 0);

```

```
addEntry(hTable, op_entry);
```

Mapping Fixed-Point Shift Left Operations to Target-Specific Implementations

- “Example: Creating a TFL Entry to Replace Shift Lefts for int16 Data” on page 29-111
- “Example: Creating a TFL Entry to Replace Fixed-Point Shift Lefts Using Net Slope” on page 29-112

You can use TFL table entries to replace the default generated code for << (shift left) operations with calls to optimized functions.

For details of the arithmetic supported for replacement of shift-left operations, see the shift left subsection of “Fixed-Point Numbers and Arithmetic” on page 29-82.

Example: Creating a TFL Entry to Replace Shift Lefts for int16 Data.

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry to replace << (shift left) operations for int16 data. In this example:

- The TFL operator entry is instantiated using the `RTW.Tf1COperationEntry` class.
- The function `setTf1COperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (shift left) and the name of the replacement function (`my_shift_left`).
- The function `getTf1ArgFromString` is called to create an int16 output argument, which is then added to the operator entry both as the first conceptual argument and the implementation return argument.
- The function `getTf1ArgFromString` is called to create an int16 input argument, which is then added to the operator entry both as the second conceptual argument and the first implementation input argument.
- The function `getTf1ArgFromString` is called to create an int8 input argument, which is then added to the operator entry both as the third conceptual argument and the second implementation input argument. This argument specifies the number of bits to shift the previous input argument.

Since the argument type is not relevant, type checking is disabled by setting the `CheckType` property to `false`.

```

hTable = RTW.TflTable;

% Create a shift left replacement for int16 data
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_SL', ...
    'Priority', 50, ...
    'ImplementationName', 'my_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

% Create int16 arg as conceptual arg 1 and implementation return
arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
op_entry.Implementation.setReturn(arg);

% Create int16 arg as conceptual arg 2 and implementation input arg 1
arg = getTflArgFromString(hTable, 'u1', 'int16');
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);

% Create int8 arg as conceptual arg 3 and implementation input arg 2
% Turn off type checking for number of bits to shift argument
arg = getTflArgFromString(hTable, 'u2', 'int8');
arg.CheckType = false;
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

```

Example: Creating a TFL Entry to Replace Fixed-Point Shift Lefts Using Net Slope. The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 29-22 to create a TFL table entry to replace << (shift left) operations for fixed-point data using a net slope. In this example:

- The TFL operator entry is instantiated using the `RTW.Tf1COperationEntryGenerator_NetSlope` class, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.
- The function `setTf1COperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (shift left), the saturation mode (saturate on overflow), the rounding mode (toward negative infinity), and the name of the replacement function (`my_fxp_shift_left`). Additionally, `NetSlopeAdjustmentFactor` and `NetFixedExponent` are used to specify the F and E parts of the net slope $F2^E$.
- The function `createAndAddConceptualArg` is called to create conceptual output and input arguments that are added to the operator entry. Each argument is specified as fixed-point and signed. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.
- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create implementation return and input arguments that are added to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).
- The function `getTf1ArgFromString` is called to create a `uint8` input argument, which is then added to the operator entry both as the third conceptual argument and the second implementation input argument. This argument specifies the number of bits to shift the previous input argument. Since the argument type is not relevant, type checking is disabled by setting the `CheckType` property to `false`.

```

hTable = RTW.Tf1Table;

% Create a fixed-point shift left replacement using a NetSlope entry
op_entry = RTW.Tf1COperationEntryGenerator_NetSlope;
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
OutWL = 32;
OutSgn = true;
setTf1COperationEntryParameters(op_entry, ...

```

```

'Key',                'RTW_OP_SL', ...
'Priority',           50, ...
'SaturationMode',   'RTW_SATURATE_ON_OVERFLOW', ...
'RoundingMode',     'RTW_ROUND_FLOOR', ...
'NetSlopeAdjustmentFactor', 1.0, ...
'NetFixedExponent', (OutFL - InFL),...
'SaturationMode',   'RTW_SATURATE_ON_OVERFLOW', ...
'RoundingMode',     'RTW_ROUND_FLOOR', ...
'ImplementationName', 'my_fxp_shift_left', ...
'ImplementationHeaderFile', 'some_hdr.h', ...
'ImplementationSourceFile', 'some_hdr.c');

% Create fixed-point arg as conceptual arg 1
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'CheckSlope',    false, ...
    'CheckBias',     false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned',      OutSgn, ...
    'WordLength',    OutWL, ...
    'FractionLength',OutFL);

% Create fixed-point arg as conceptual arg 2
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'CheckSlope',    false, ...
    'CheckBias',     false, ...
    'DataTypeMode', 'Fixed-point: binary point scaling', ...
    'IsSigned',      InSgn, ...
    'WordLength',    InWL, ...
    'FractionLength',InFL);

% Create implementation return arg
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'IsSigned',      OutSgn, ...
    'WordLength',    OutWL, ...

```

```

        'FractionLength', 0);

% Create implementation input arg 1
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      InSgn, ...
    'WordLength',    InWL, ...
    'FractionLength', 0);

% Create uint8 arg as conceptual arg 3 and implementation input arg 2
% Turn off type checking for number of bits to shift argument
arg = getTf1ArgFromString(hTable, 'u2', 'uint8');
arg.CheckType = false;
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

```

Remapping Operator Outputs to Implementation Function Input Positions

If you need your generated code to meet a specific coding pattern or you want more flexibility, for example, to further improve performance, you have the option of remapping operator outputs to input positions in an implementation function argument list.

Note Remapping outputs to implementation function inputs is supported only for operator replacement.

For example, for a sum operation, the build process might generate code similar to the following:

```
rtY.Out1 = u8_add_u8_u8(rtU.In1, rtU.In2);
```

If you remap the output to the first input, the build process generates code similar to the following:

```
uint8_T rtb_Add8;
```

```
u8_add_u8_u8(&rtb_Add8, rtU.In1, rtU.In2);
rtY.Out1 = rtb_Add8;
```

To remap an operator output to an implementation function input for an existing TFL operator replacement entry, you modify the TFL table definition file as follows:

- 1 In the `setTflCOperationEntryParameters` function call for the operator replacement, specify the `SideEffects` parameter as `true`.
- 2 When defining the implementation function return, create a new void output argument, for example, `y2`.
- 3 When defining the implementation function arguments, set the operator output argument (for example, `y1`) as an additional input argument, marking its `IOType` as output, and make its type a pointer type.

For example, the following TFL table definition file for a sum operation has been modified to remap operator output `y1` as the first function input argument. The modified lines of code are shown in **bold** type. (This definition file generated the example remap code shown above.)

```
function hTable = tfl_table_add_uint8
%TFL_TABLE_ADD_UINT8 - Describe operator entry for a Target Function Library table.

hTable = RTW.TflTable;

% Create entry for addition of built-in uint8 data type
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                'RTW_OP_ADD', ...
    'Priority',            90, ...
    'ImplementationName', 'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c', ...
    SideEffects ,        true );

arg = getTflArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
```



```

arg = getTflArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(op_entry, arg );

arg = getTflArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(op_entry, arg );

% Create new void output y2
arg = getTflArgFromString(hTable, y2 , void );
arg.IOType = RTW_IO_OUTPUT ;
op_entry.Implementation.setReturn(arg);

% Set y1 as first input arg, mark IOType as output, and use pointer type
arg=getTflArgFromString(hTable, y1 , uint8* );
arg.IOType = RTW_IO_OUTPUT ;
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u1', 'uint8');
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u2', 'uint8');
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

```

Refining TFL Matching and Replacement Using Custom TFL Table Entries

- “Example: Customizing TFL Matching and Replacement for Operators” on page 29-119
- “Example: Customizing TFL Matching and Replacement for Functions” on page 29-128

During code generation for your model, the TFL replacement capability uses

- Preset match criteria to identify math functions and operators for which target-specific implementations should replace default implementations
- Preset replacement function signatures

However, preset match criteria and preset replacement function signatures might not be sufficient for all function and operator replacement needs. For example,

- You might want to replace an operator with a particular fixed-point implementation function only when fraction lengths are within a particular range.
- When a match is made, you might want to modify your replacement function signature based on compile-time information, such as passing fraction-length values into the function.

When you need to add extra logic into the TFL matching and replacement process, you can create custom TFL table entries. Custom entries allow you to specify additional match criteria and/or modify the replacement function signature to meet your application needs.

To create a custom TFL table entry, you perform the following steps:

- 1** Create a custom TFL entry class, derived from either `RTW.Tf1COperationEntryML` (for operator replacement) or `RTW.Tf1CFunctionEntryML` (for function replacement).
- 2** In your derived class, implement a `do_match` method with a fixed preset signature as a MATLAB function. In your `do_match` method, you can provide either or both of the following customizations for use by TFL table entries that instantiate the class:
 - a** Add any additional match criteria not provided by the base class. The base class provides a match based on argument number, argument name, signedness, word size, slope (if not wildcarded), bias (if not wildcarded), math modes such as saturation and rounding, and operator or function key. For example, you can accept a match only when additional size or range conditions are met.
 - b** Modify the implementation signature by adding additional arguments or setting constant input argument values. For example, you can inject a constant value, such as an input's scaling value, as an additional argument to the replacement function.
- 3** Create TFL table entries that instantiate your custom TFL entry class.

- 4 Register a TFL containing the TFL table entries. The registered TFL is then available for selection in the **Interface** pane of the Simulink Configuration Parameters dialog box.

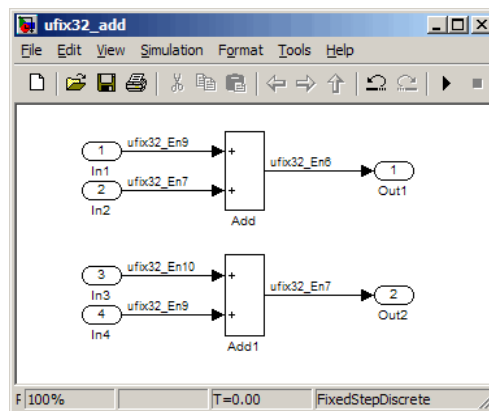
During code generation, the TFL matching process first tries to match function or operator call sites with the base class of your derived entry class. If a match is found, the software calls your `do_match` method to execute your additional match logic (if any) and your replacement function customizations (if any).

The following sections provide examples of creating custom TFL table entries to refine matching and replacement for operators and functions. For more examples, see the TFL demos page, including the demo model `rtwdemo_tflcustomentry`.

Example: Customizing TFL Matching and Replacement for Operators

This example demonstrates how to use custom TFL table entries to refine the matching and replacement logic for operators. In this example, a fixed-point addition replacement needs to be modified such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

- 1 To exercise the custom TFL table entries created in this example, create an ERT-based model with one or more unsigned 32-bit fixed-point addition operations, such as the following:



For the purposes of this example, in the block parameters for both Add blocks, set **Integer rounding mode** to Floor and select the option **Saturate on integer overflow**.

- 2 Create a class folder using the name of your derived class, such as @Tf1CustomOperationEntry. Make sure the class folder is in the MATLAB search path or in the current working folder.
- 3 In the class folder, create and save the following class definition file, Tf1CustomOperationEntry.m. This file defines the class Tf1CustomOperationEntry, which is derived from the base class RTW.Tf1COperationEntryML

The derived class defines a do_match method . In the do_match method signature,

- ent is the return handle, which is returned either as empty (indicating that the match failed) or as a Tf1COperationEntry handle.
- hThis is the handle to this object.
- hCSO is a handle to an object created by the code generator for the purpose of querying the TFL for a replacement.
- The remaining arguments are the number of bits for various data types of the current target.

The purpose of the do_match method is to add any additional match criteria not provided by the base class and make any desired modifications to the implementation signature. In this case, the do_match method can rely on the base class for checking word size and signedness, and additionally only needs to match the number of conceptual arguments to the value 3 (two inputs and one output) and the bias for each argument to the value 0. If a match is made, the method sets the return handle, removes slope and bias wildcarding from the conceptual arguments (since the match is for specific slope and bias values), and writes fraction-length values for the inputs and output into replacement function arguments 3, 4, and 5.

Note The three additional implementation function arguments for passing fraction lengths can be created and added either here in the class definition or in each TFL table entry definition that instantiates this class. In this example, the arguments are created and added in a TFL table definition file and set to specific values in the class definition code. For an example of creating and adding additional implementation function arguments in a class definition, see “Example: Customizing TFL Matching and Replacement for Functions” on page 29-128.

```

classdef TflCustomOperationEntry < RTW.Tf1COperationEntryML
    methods
        function ent = do_match(hThis, ...
            hCSO, ... %#ok
            targetBitPerChar, ... %#ok
            targetBitPerShort, ... %#ok
            targetBitPerInt, ... %#ok
            targetBitPerLong) %#ok
            % DO_MATCH - Create a custom match function. The base class
            % checks the types of the arguments prior to calling this
            % method. This will check additional data and perhaps modify
            % the implementation function.

            % The base class checks word size and signedness. Slopes and biases
            % have been wildcarded, so the only additional checking needed is
            % to make sure the biases are zero and that there are only three
            % conceptual arguments (one output, two inputs)

            ent = []; % default the return to empty, indicating the match failed.

            if length(hCSO.ConceptualArgs) == 3 && ...
                hCSO.ConceptualArgs(1).Type.Bias == 0 && ...
                hCSO.ConceptualArgs(2).Type.Bias == 0 && ...
                hCSO.ConceptualArgs(3).Type.Bias == 0

                % Need to modify the default implementation. Since this is a
                % generator entry, a concrete entry is created using this entry
                % as a template. The type of entry being created is a standard
                % Tf1COperationEntry. Using the standard operation entry is

```

```

% sufficient since it contains all the necessary information and
% a custom match function will no longer be needed.
ent = RTW.Tf1COperationEntry(hThis);

% Since this entry is modifying the implementation for specific
% fraction-length values (arguments 3, 4, and 5), the conceptual argument
% wildcards must be removed (the wildcards were inherited from the
% generator when it was used as a template for the concrete entry).
% This concrete entry is now for a specific slope and bias
% (not for any slope and bias). The hCSO holds the correct
% slope and bias values (created by the code generator).
for idx=1:3
    ent.ConceptualArgs(idx).CheckSlope = true;
    ent.ConceptualArgs(idx).CheckBias = true;

    % Set the specific Slope and Biases
    ent.ConceptualArgs(idx).Type.Slope = hCSO.ConceptualArgs(idx).Type.Slope;
    ent.ConceptualArgs(idx).Type.Bias = 0;
end

% Set the fraction-length values in the implementation function.
ent.Implementation.Arguments(3).Value = ...
    -1.0*hCSO.ConceptualArgs(2).Type.FixedExponent;
ent.Implementation.Arguments(4).Value = ...
    -1.0*hCSO.ConceptualArgs(3).Type.FixedExponent;
ent.Implementation.Arguments(5).Value = ...
    -1.0*hCSO.ConceptualArgs(1).Type.FixedExponent;
end
end
end
end
end

```

Exit the class folder and return to the previous working folder.

- 4** Create and save the following TFL table definition file, `tfl_table_custom_add_ufix32.m`. This file defines a TFL table containing a single operator entry, a TFL entry generator for unsigned 32-bit fixed-point addition operations, with arbitrary fraction-length values on the inputs and the output. This entry instantiates the derived class from the previous step, `Tf1CustomOperationEntry`.

Note

- If you want to replace all word sizes and signedness attributes (not just 32-bit and unsigned), you can use the same derived class, but not the same TFL entry, because the `WordLength` and `IsSigned` arguments cannot be wildcarded. For example, to support `uint8`, `int8`, `uint16`, `int16`, and `int32`, you would need to add five other distinct TFL entries. Similarly, if you wanted to use different implementation functions for saturation and rounding modes other than overflow and round to floor, you would need to add TFL entries for those match permutations.
- This table entry creates and adds three implementation arguments to hold the fraction-length values for the inputs and output. Alternatively, this table entry could omit those argument definitions and instead the `do_match` method of the derived class `TflCustomOperationEntry` could create and add the three implementation arguments. In particular, you should use the alternative approach when the number of additional implementation arguments required might vary based on compile-time information.

```
function hTable = tfl_table_custom_add_ufix32

hTable = RTW.TflTable;

%% Add TflCustomOperationEntry
op_entry = TflCustomOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
    'Key',          'RTW_OP_ADD', ...
    'Priority',     30, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingMode', 'RTW_ROUND_FLOOR', ...
    'ImplementationName', 'myFixptAdd', ...
    'ImplementationHeaderFile', 'myFixptAdd.h', ...
    'ImplementationSourceFile', 'myFixptAdd.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',          'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'CheckSlope',   false, ...
```

```

        'CheckBias', false, ...
        'DataType', 'Fixed', ...
        'Scaling', 'BinaryPoint', ...
        'IsSigned', false, ...
        'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'Scaling', 'BinaryPoint', ...
    'IsSigned', false, ...
    'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u2', ...
    'IOType', 'RTW_IO_INPUT', ...
    'CheckSlope', false, ...
    'CheckBias', false, ...
    'DataType', 'Fixed', ...
    'Scaling', 'BinaryPoint', ...
    'IsSigned', false, ...
    'WordLength', 32);

% Specify replacement function signature
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'IsSigned', false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'IsSigned', false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

```



```

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',      'u2', ...
    'IOType',    'RTW_IO_INPUT', ...
    'IsSigned',  false, ...
    'WordLength', 32, ...
    'FractionLength', 0);

% Add 3 fraction-length args. Actual values will be set during code generation.
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumericConstant', ...
    'Name',      'fl_in1', ...
    'IOType',    'RTW_IO_INPUT', ...
    'IsSigned',  false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',     0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumericConstant', ...
    'Name',      'fl_in2', ...
    'IOType',    'RTW_IO_INPUT', ...
    'IsSigned',  false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',     0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumericConstant', ...
    'Name',      'fl_out', ...
    'IOType',    'RTW_IO_INPUT', ...
    'IsSigned',  false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',     0);

addEntry(hTable, op_entry);

```

- 5** Optionally, perform a quick check of the validity of the function entry by invoking the table definition file at the MATLAB command line (>> `tbl = tf1_table_custom_add_ufix32`) and by viewing it in the TFL Viewer (>> `RTW.viewTf1(tf1_table_custom_add_ufix32)`). For more information

about validating TFL tables, see “Examining and Validating Function Replacement Tables” on page 29-140.

- 6 Create and save the following TFL registration file, which references the `tfl_table_custom_add_ufix32` table.

The file specifies that the TFL to be registered is named 'Custom TFL Operator Entry Example' and consists of `tfl_table_custom_add_ufix32`, with the default ANSI math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION

% Local function to define a TFL containing tfl_table_custom_add_ufix32
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties
thisTfl.Name = 'Custom TFL Operator Entry Example';
thisTfl.Description = 'Demonstration of custom match for operator replacement';
thisTfl.TableList = {'tfl_table_custom_add_ufix32'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`;))

For more information about registering TFLs with Simulink or Embedded MATLAB Coder software, see “Registering Target Function Libraries” on page 29-149.

- 7 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open the model you created in step 1 and navigate to the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including Custom TFL Operator Entry Example.

- 8 Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Real-Time Workshop** pane, select the **Generate code only** option, and generate code for the model.
- 9 Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click either Add block and select **Real-Time Workshop > Navigate to Code**. This selection highlights the Sum block code within the model step function in the `model.c` file. As shown below, the default implementation code for the unsigned 32-bit fixed-point addition operation has been replaced with `myFixptAdd`, and the three additional fraction-length arguments are present.

```

/* Model step function */
void ufix32_add_step(void)
{
    /* Output: '<Root>/Out1' incorporates:
    * Inport: '<Root>/In1'
    * Inport: '<Root>/In2'

```

```

* Sum: '<Root>/Add1'
*/
ufix32_add_Y.Out1 = myFixptAdd(ufix32_add_U.In1, ufix32_add_U.In2, 9U, 7U, 6U);

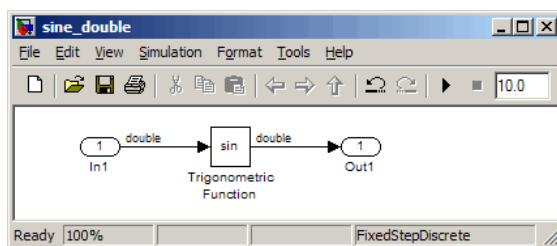
/* Outport: '<Root>/Out2' incorporates:
* Inport: '<Root>/In3'
* Inport: '<Root>/In4'
* Sum: '<Root>/Add1'
*/
ufix32_add_Y.Out2 = myFixptAdd(ufix32_add_U.In3, ufix32_add_U.In4, 10U, 9U, 7U);
}

```

Example: Customizing TFL Matching and Replacement for Functions

This example demonstrates how to use custom TFL table entries to refine the matching and replacement logic for functions. In this example, a sine function replacement needs to be modified, only if the integer size on the current target platform is 32 bits, such that the implementation function passes in a degrees-versus-radians flag as an input argument.

- 1 To exercise the custom TFL table entries created in this example, create an ERT-based model with a sine function block, such as the following:



For the purposes of this example, in the import block parameters, set the signal **Data type** to double. Also, if the target platform selected for your model on the **Hardware Implementation** pane of the Configuration Parameters dialog box supports an integer size other than 32, you should either temporarily select a target platform with a 32-bit integer size, or modify the code in this example to match the integer size of your target platform.

- 2 Create a class folder using the name of your derived class, such as `@TflCustomFunctionEntry`. Make sure the class folder is in the MATLAB search path or in the current working folder.
- 3 In the class folder, create and save the following class definition file, `TflCustomFunctionEntry.m`. This file defines the class `TflCustomFunctionEntry`, which is derived from the base class `RTW.TflCFunctionEntryML`

The derived class defines a `do_match` method. In the `do_match` method signature,

- `ent` is the return handle, which is returned either as empty (indicating that the match failed) or as a `TflCFunctionEntry` handle.
- `hThis` is the handle to this object.
- `hCSO` is a handle to an object created by the code generator for the purpose of querying the TFL for a replacement.
- The remaining arguments are the number of bits for various data types of the current target.

The purpose of the `do_match` method is to add any additional match criteria not provided by the base class and make any desired modifications to the implementation signature. In this case, the `do_match` method only needs to match `targetBitPerInt`, representing the number of bits in the `C int` data type for the current target, to the value 32. If a match is made, the method sets the return handle and creates and adds an input argument, representing whether units are expressed as degrees or radians, to the replacement function signature.

Note Alternatively, the additional implementation function argument for passing a units flag could be created and added in each TFL table definition file that instantiates this class. In that case, this class definition code would not create the argument and would only set its value. For an example of creating and adding additional implementation function arguments in a table definition file, see “Example: Customizing TFL Matching and Replacement for Operators” on page 29-119.

```

classdef TflCustomFunctionEntry < RTW.TflCFunctionEntryML
    methods
        function ent = do_match(hThis, ...
            hCSO, ... %#ok
            targetBitPerChar, ... %#ok
            targetBitPerShort, ... %#ok
            targetBitPerInt, ... %#ok
            targetBitPerLong) %#ok
            % DO_MATCH - Create a custom match function. The base class
            % checks the types of the arguments prior to calling this
            % method. This will check additional data and perhaps modify
            % the implementation function.

            ent = []; % default the return to empty, indicating the match failed.

            % Match sine function only if the target int size is 32 bits
            if targetBitPerInt == 32
                % Need to modify the default implementation, starting from a copy
                % of the standard TflCFunctionEntry.
                ent = RTW.TflCFunctionEntry(hThis);

                % If the target int size is 32 bits, the implementation function
                % takes an additional input flag argument indicating degrees vs.
                % radians. The additional argument can be created and added either
                % in the TFL table definition file that instantiates this class, or
                % here in the class definition, as follows:
                createAndAddImplementationArg(ent, 'RTW.TflArgNumericConstant', ...
                    'Name',          'u2', ...
                    'IsSigned',      true, ...
                    'WordLength',    32, ...
                    'FractionLength', 0, ...
                    'Value',         1);
            end
        end
    end
end
end
end

```

Exit the class folder and return to the previous working folder.

- 4** Create and save the following TFL table definition file, `tfl_table_custom_sinfcn_double.m`. This file defines a TFL table containing a function table entry for sine with double input and output. This entry instantiates the derived class from the previous step, `TflCustomFunctionEntry`.

```
function hTable = tfl_table_custom_sinfcn_double

hTable = RTW.TflTable;

%% Add TflCustomFunctionEntry
fcn_entry = TflCustomFunctionEntry;
setTflCFunctionEntryParameters(fcn_entry, ...
    'Key',                'sin', ...
    'Priority',            30, ...
    'ImplementationName', 'mySin', ...
    'ImplementationHeaderFile', 'mySin.h', ...
    'ImplementationSourceFile', 'mySin.c');

createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
    'Name',                'y1', ...
    'IOType',              'RTW_IO_OUTPUT', ...
    'DataTypeMode',       'double');

createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
    'Name',                'u1', ...
    'IOType',              'RTW_IO_INPUT', ...
    'DataTypeMode',       'double');

% TflCustomFunctionEntry class do_match method will create and add
% an implementation function argument during code generation if
% the supported integer size on the current target is 32 bits.
copyConceptualArgsToImplementation(fcn_entry);

addEntry(hTable, fcn_entry);
```

- 5** Optionally, perform a quick check of the validity of the function entry by invoking the table definition file at the MATLAB command line (`>> tbl = tfl_table_custom_sinfcn_double`) and by viewing it in the TFL Viewer (`>> RTW.viewTfl(tfl_table_custom_sinfcn_double)`). For more

information about validating TFL tables, see “Examining and Validating Function Replacement Tables” on page 29-140.

- 6 Create and save the following TFL registration file, which references the `tfl_table_custom_sinfcn_double` table.

The file specifies that the TFL to be registered is named 'Custom TFL Function Entry Example' and consists of `tfl_table_custom_sinfcn_double`, with the default ANSI math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION

% Local function to define a TFL containing tfl_table_custom_sinfcn_double
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties
thisTfl.Name = 'Custom TFL Function Entry Example';
thisTfl.Description = 'Demonstration of custom match for function replacement';
thisTfl.TableList = {'tfl_table_custom_sinfcn_double'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`;))

For more information about registering TFLs with Simulink or Embedded MATLAB Coder software, see “Registering Target Function Libraries” on page 29-149.

- 7 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open the model you created in step 1 and navigate to the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including Custom TFL Function Entry Example.

- 8 Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Real-Time Workshop** pane, select the **Generate code only** option, and generate code for the model.
- 9 Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the sine block and select **Real-Time Workshop > Navigate to Code**. This selection highlights the sine block code within the model step function in the `model.c` file. As shown below, the default implementation code for the sine function has been replaced with `mySin`, and the additional units flag input argument is present.

```

/* Model step function */
void sine_double_step(void)
{
    /* Output: '<Root>/Out1' incorporates:
    * Inport: '<Root>/In1'
    * Trigonometry: '<Root>/Trigonometric Function'
    */

```

```
sine_double_Y.Out1 = mySin(sine_double_U.In1, 1);  
}
```

Note Optionally, you can change the current target for the model such that the supported integer size is not 32 bits, and then regenerate code. In that case, the custom match is not made, and the additional input argument does not appear in the generated code for the sine block.

Specifying Build Information for Function Replacements

- “Functions for Specifying Table Entry Build Information” on page 29-134
- “Using RTW.copyFileToBuildDir to Copy Files to the Build Folder” on page 29-135
- “RTW.copyFileToBuildDir Examples” on page 29-136

Functions for Specifying Table Entry Build Information

As you create TFL table entries for function or operator replacement, you specify the header and source file information for each function implementation using one of the following:

- The arguments `ImplementationHeaderFile`, `ImplementationHeaderPath`, `ImplementationSourceFile`, and `ImplementationSourcePath` to `setTflCFunctionEntryParameters` or `setTflCOperationEntryParameters`
- The `headerFile` argument to `registerCFunctionEntry`, `registerCPPFunctionEntry`, or `registerCPromotableMacroEntry`

Also, each table entry can specify additional header files, source files, and object files to be included in model builds whenever the TFL table entry is matched and used to replace a function or operator in generated code. To add an additional header file, source file, or object file, use the following TFL table creation functions.

Function	Description
<code>addAdditionalHeaderFile</code>	Add additional header file to array of additional header files for TFL table entry
<code>addAdditionalIncludePath</code>	Add additional include path to array of additional include paths for TFL table entry
<code>addAdditionalLinkObj</code>	Add additional link object to array of additional link objects for TFL table entry
<code>addAdditionalLinkObjPath</code>	Add additional link object path to array of additional link object paths for TFL table entry
<code>addAdditionalSourceFile</code>	Add additional source file to array of additional source files for TFL table entry
<code>addAdditionalSourcePath</code>	Add additional source path to array of additional source paths for TFL table entry

For function descriptions and examples, see the function reference pages in the Real-Time Workshop Embedded Coder reference documentation.

Using `RTW.copyFileToBuildDir` to Copy Files to the Build Folder

If a TFL table entry uses header, source, or object files that reside in external directories, and if the table entry is matched and used to replace a function or operator in generated code, the external files will need to be copied to the build folder before the generated code is built. The `RTW.copyFileToBuildDir` function can be invoked after code generation to copy the table entry's specified header file, source file, additional header files, additional source files, and additional link objects to the build folder. The copied files are then available for use in the build process.

To direct that a table entry's external files should be copied to the build folder after code generation, specify the argument `'RTW.copyFileToBuildDir'` to the `genCallback` parameter of the TFL function that you use to set the table entry parameters, among the following:

- `registerCFunctionEntry`
- `registerCPPFunctionEntry`

- registerCPromotableMacroEntry
- setTf1CFunctionEntryParameters
- setTf1COperationEntryParameters

RTW.copyFileToBuildDir Examples

The following example defines a table entry for an optimized multiplication function that takes signed 32-bit integers and returns a signed 32-bit integer, taking saturation into account. Multiplications in the generated code will be replaced with calls to your optimized function. Your optimized function resides in an external folder and must be copied into the build folder to be compiled and linked into the application.

The multiplication table entry specifies the source and header file names as well as their full paths. To request the copy to be performed, the table entry specifies the argument 'RTW.copyFileToBuildDir' to the genCallback parameter of the setTf1COperationEntryParameters function. In this example, the header file s32_mul.h contains an inlined function that invokes assembly functions contained in s32_mul.s. If the table entry is matched and used to generate code, the RTW.copyFileToBuildDir function will copy the specified source and header files into the build folder.

```
function hTable = make_my_tfl_table

hTable = RTW.Tf1Table;

op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingMode', 'RTW_ROUND_UNSPECIFIED', ...
    'ImplementationName', 's32_mul_s32_s32_sat', ...
    'ImplementationHeaderFile', 's32_mul.h', ...
    'ImplementationSourceFile', 's32_mul.s', ...
    'ImplementationHeaderPath', {fullfile('${MATLAB_ROOT}','tfl')}, ...
    'ImplementationSourcePath', {fullfile('${MATLAB_ROOT}','tfl')}, ...
    'GenCallback', 'RTW.copyFileToBuildDir');
```

```

.
.
addEntry(hTable, op_entry);

```

The following example shows the use of the `addAdditional*` functions along with `RTW.copyFileToBuildDir`.

```

hTable = RTW.Tf1Table;

% Path to external source, header, and object files
libdir = fullfile('$MATLAB_ROOT','..', '..', 'lib');

op_entry = RTW.Tf1COperationEntry;
setTf1COperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_SATURATE_UNSPECIFIED', ...
    'RoundingMode', 'RTW_ROUND_UNSPECIFIED', ...
    'ImplementationName', 's32_add_s32_s32', ...
    'ImplementationHeaderFile', 's32_add_s32_s32.h', ...
    'ImplementationSourceFile', 's32_add_s32_s32.c'...
    'GenCallback', 'RTW.copyFileToBuildDir');

addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));
addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));

.
.
.
addEntry(hTable, op_entry);

```

Adding Target Function Library Reserved Identifiers

The 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 the Real-Time Workshop software or C programming and should not be used in Simulink models as identifiers or function names. Real-Time Workshop reserved keywords

include many TFL identifiers, the majority of which are function names, such as `acos`. To view a list of reserved identifiers for the TFL that you are using to generate code, call the MATLAB function `RTW.TargetRegistry.getInstance.getTflReservedIdentifiers`, 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)')
```

For more information, see “Real-Time Workshop Target Function Library Keywords” in the Real-Time Workshop documentation.

In a TFL table, each function implementation name defined by a table entry is registered as a reserved identifier. You can register additional reserved identifiers for the table on a per-header-file basis. Providing additional reserved identifiers can help prevent duplicate symbols and other identifier-related compile and link issues.

To register additional TFL reserved identifiers, use the following function.

Function	Description
<code>setReservedIdentifiers</code>	Register specified reserved identifiers to be associated with TFL table

You can register up to four reserved identifier structures in a TFL table. One set of reserved identifiers can be associated with an arbitrary TFL, while the other three (if present) must be associated with ANSI, ISO¹⁵, or GNU¹⁶ libraries. The following example shows a reserved identifier structure that specifies two identifiers and the associated header file.

```
d{1}.LibraryName = 'ANSI';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

The specified identifiers are added to the reserved identifiers collection and honored during the Real-Time Workshop build procedure. For more

15. ISO® is a registered trademark of the International Organization for Standardization.

16. GNU® is a registered trademark of the Free Software Foundation.

information and examples, see `setReservedIdentifiers` in the Real-Time Workshop Embedded Coder reference documentation.

Examining and Validating Function Replacement Tables

In this section...

“Overview of Function Replacement Table Validation” on page 29-140

“Invoking the Table Definition File” on page 29-140

“Using the Target Function Library Viewer to Examine Your Table” on page 29-141

“Using the Target Function Library Viewer to Examine Registered TFLs” on page 29-142

“Tracing Code Generated Using Your Target Function Library” on page 29-144

“Examining TFL Cache Hits and Misses” on page 29-145

Overview of Function Replacement Table Validation

After you create a target function library (TFL) table containing your function replacement entries, but before you deploy production TFLs containing your table for general use in building models, you can use various techniques to examine and validate the TFL table entries. These include:

- Invoking the table definition file
- Using the TFL Viewer at various stages of TFL development to examine TFLs, tables, and entries
- Tracing code generated from models for which your TFL is selected
- Examining TFL cache hits and misses logged during code generation

Invoking the Table Definition File

Immediately after creating or modifying a table definition file (as described in “Creating Function Replacement Tables” on page 29-16), you should invoke it at the MATLAB command line. This invocation serves as a check of the validity of your table entries. For example,

```
>> tbl = tf1_table_sinfcn
```

```
tbl =
```



```

RTW.Tf1Table
    Version: '1.0'
    AllEntries: [2x1 RTW.Tf1CFunctionEntry]
    ReservedSymbols: []
    StringResolutionMap: []
>>

```

Any errors found during the invocation are displayed. In the following example, a typo in a data type name is detected and displayed.

```

>> tbl = tf1_table_sinfcn
??? RTW_CORE:tf1:Tf1Table: Unsupported data type, 'dooble'.

Error in ==> tf1_table_sinfcn at 7
hTable.registerCFunctionEntry(100, 1, 'sin', 'dooble', 'sin_dbl', ...

>>

```

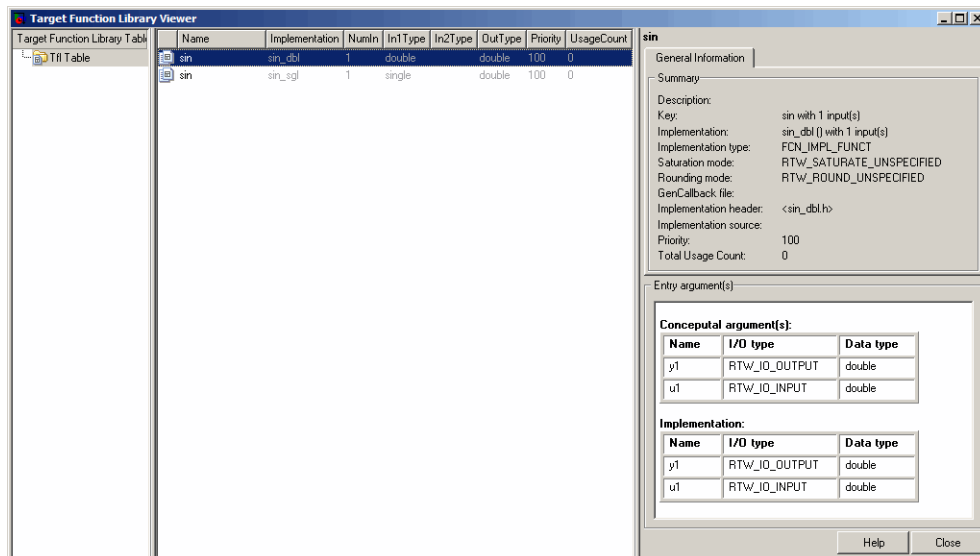
Using the Target Function Library Viewer to Examine Your Table

After creating or modifying a table definition file, as a further check of your table entries, you should use the TFL Viewer to display and examine your table. Invoke the TFL Viewer using the following form of the MATLAB command `RTW.viewTf1`:

```
RTW.viewTf1(table-name)
```

For example,

```
>> RTW.viewTf1(tf1_table_sinfcn)
```



Select entries in your table and verify that the graphical display of the contents of your table meets your expectations. Common problems that can be detected at this stage include:

- Incorrect argument order
- Conceptual argument naming that does not match the naming convention used by the code generation process
- Incorrect relative priority of entries within the table (highest priority is 0, and lowest priority is 100).

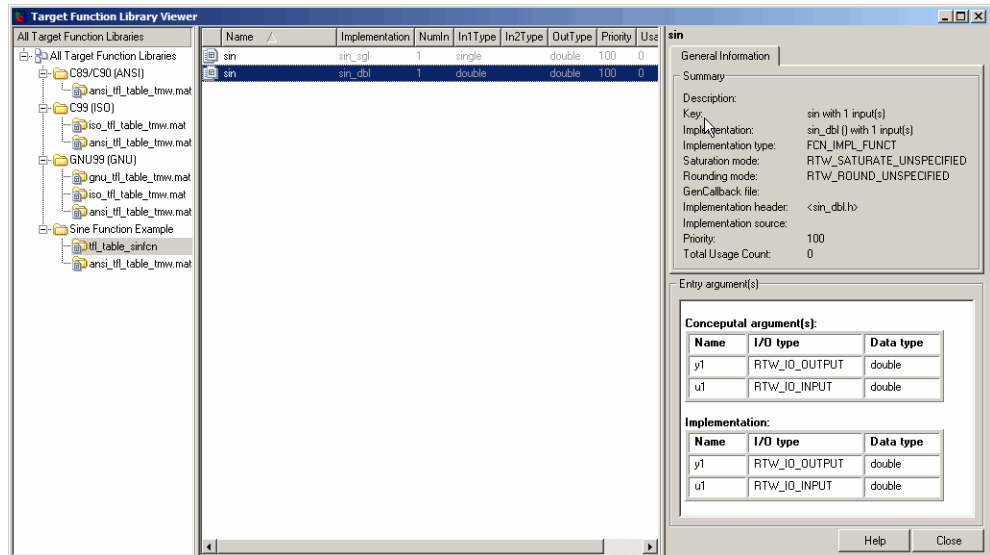
For more information about the TFL Viewer, see “Using the Target Function Library Viewer” in the Real-Time Workshop documentation.

Using the Target Function Library Viewer to Examine Registered TFLs

After you register a TFL that includes your function replacement table (as described in “Registering Target Function Libraries” on page 29-149), you should use the TFL Viewer to verify that your TFL was properly registered and to examine the TFL and the tables it contains. Invoke the TFL Viewer

using the MATLAB command `RTW.viewTf1` with no arguments. This command displays all TFLs registered in the current Simulink session. For example:

```
>> RTW.viewTf1
```



If your TFL is not displayed,

- There may be an error in your TFL registration file.
- You may need to refresh the TFL registration information by issuing the MATLAB command `sl_refresh_customizations` or, for an Embedded MATLAB Coder TFL registration, using the command `RTW.TargetRegistry.getInstance('reset')`.

If your TFL is displayed, select the TFL and examine and compare its tables, including their relative order. Common problems that can be detected at this stage include

- Incorrect relative order of tables in the library (tables are displayed in search order)
- Table entry problems as listed in the previous section

For more information about the TFL Viewer, see “Using the Target Function Library Viewer” in the Real-Time Workshop documentation.

Tracing Code Generated Using Your Target Function Library

After you register a TFL that includes your function replacement tables, you should use the TFL to generate code and verify that you are obtaining the function or operator replacement that you expect. For example, the following approach uses model-to-code highlighting to trace a specific expected replacement.

- 1** Open a ERT-based model for which you anticipate that a function or operator replacement should occur.
- 2** Select your TFL in the **Target function library** drop-down list on the **Interface** pane of the Configuration Parameters dialog box.
- 3** Go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**.
- 4** Go to the **Real-Time Workshop** pane, select the **Generate code only** option, and generate code for the model.
- 5** Go to the model window and use model-to-code highlighting to trace the code generated using your TFL. For example, right-click a block that you expect to have generated a function or operator replacement and select **Real-Time Workshop > Navigate to Code**. This selection highlights the applicable generated function code within `sinefcn.c` as shown in HTML code generation report.

```
21
22 /* Real-time model */
23 RT_MODEL_sinefcn sinefcn_M;
24 RT_MODEL_sinefcn *sinefcn_M = &sinefcn_M;
25
26 /* Model step function */
27 void sinefcn_step(void)
28 {
29     /* Outport: '<Root>/Out1' incorporates:
30      * Inport: '<Root>/In1'
31      * Trigonometry: '<Root>/Trigonometric Function'
32      */
33     sinefcn_Y.Out1 = sin_dbl(sinefcn_U.In1);
34 }
35
36 /* Model initialize function */
37 void sinefcn_initialize(void)
38 {
39     /* Registration code */
```

Inspect the generated code and see if the function or operator replacement occurred as you expected.

Note If a function or operator was not replaced as you expected, it means that a call site request was not matched as you intended by your table entry attributes. Either a higher-priority (lower priority value) match was used or no match was found. You can analyze the TFL table entry matching behavior by using the following resources together:

- TFL Viewer, as described in “Using the Target Function Library Viewer to Examine Your Table” on page 29-141 and “Using the Target Function Library Viewer to Examine Registered TFLs” on page 29-142
- HTML code generation reports, with bidirectional tracing including model-to-code highlighting
- Statistics for TFL cache hits and misses logged during code generation, as described in “Examining TFL Cache Hits and Misses” on page 29-145

Examining TFL Cache Hits and Misses

Target function library (TFL) replacement may behave differently than you expect in some cases. To verify that you are obtaining the function or

operator replacement that you expect, you first inspect the generated code, as described in “Tracing Code Generated Using Your Target Function Library” on page 29-144.

To analyze replacement behavior, in addition to referencing the generated code and examining your TFL tables in the TFL Viewer, you can view the TFL cache hits and misses logged during the most recent code generation session. This approach provides information on what data types and attributes should be registered in order to achieve the desired replacement.

To display the TFL cache hits and misses logged during the most recent code generation session, you specify the model parameter `TargetFcnLibHandle` in a `get_param` call, as follows:

```
>> tfl=get_param('model', 'TargetFcnLibHandle')
```

The resulting display includes the following fields:

Field	Description
HitCache	Table containing function entries that were successfully matched during a code generation session. These entries represent function implementations that should appear in the generated code.
MissCache	Table containing function entries that failed to match during a code generation session. These entries are created by the code generation process for the purpose of querying the TFL to locate a registered implementation. If there is a registered implementation that you feel should have been used in the generated code and was not, examining the MissCache for entries that are similar but did not match can help you locate discrepancies in a conceptual argument list or in table entry attributes.

Note You also can view cache hits and misses in the TFL Viewer, using the TFL handle returned by the `get_param` call. For example:

```
>> tf1=get_param('model', 'TargetFcnLibHandle')
>> RTW.viewTfl(tf1)
```

This opens the TFL viewer. You can then examine the cache hits and misses by clicking on the entries under those caches.

In the following example, the most recent code generation session logged one cache hit and zero cache misses. You can examine the logged `HitCache` entry using its table index.

```
>> a=get_param( sinefcn , TargetFcnLibHandle )

a =

RTW.TflControl
    Version: '1.0'
    HitCache: [1x1 RTW.TflCFunctionEntry]
    MissCache: [0x1 handle]
    TLCCallList: [0x1 handle]
    TflTables: [2x1 RTW.TflTable]

>> a.HitCache(1)

ans =

RTW.TflCFunctionEntry
    Key: 'sin'
    Priority: 100
    ConceptualArgs: [2x1 RTW.TflArgNumeric]
    Implementation: [1x1 RTW.CImplementation]
    RTWmakecfgLibName: ''
    GenCallback: ''
    GenFileName: ''
    SaturationMode: 'RTW_SATURATE_UNSPECIFIED'
    RoundingMode: 'RTW_ROUND_UNSPECIFIED'
    AcceptExprInput: 1
```

```
SideEffects: 0
UsageCount: 2
SharedUsageCount: 0
Description: ''
ImplType: 'FCN_IMPL_FUNCT'
AdditionalHeaderFiles: {0x1 cell}
AdditionalIncludePaths: {0x1 cell}
AdditionalSourceFiles: {0x1 cell}
AdditionalSourcePaths: {0x1 cell}
AdditionalLinkObjs: {0x1 cell}
AdditionalLinkObjsPaths: {0x1 cell}
```

```
>>
```


Registering Target Function Libraries

In this section...

“Overview of TFL Registration” on page 29-149

“Using the `sl_customization` API to Register a TFL with Simulink Software” on page 29-150

“Using the `rtwTargetInfo` API to Register a TFL with Embedded MATLAB Coder Software” on page 29-154

“Registering Multiple TFLs” on page 29-156

Overview of TFL Registration

After you define function and operator replacements in a target function library (TFL) table definition file, your table can be included in a TFL that you register either with Simulink software or with Embedded MATLAB Coder software. When a TFL is registered, it appears in the **Target function library** drop-down list on the **Interface** pane of either the Simulink Configuration Parameters dialog box or the Embedded MATLAB Coder Real-Time Workshop dialog box. You can select it from the **Target function library** drop-down list for use in code generation.

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

To register TFLs with Embedded MATLAB Coder software, use the Embedded MATLAB Coder customization file `rtwTargetInfo.m`. This file is a mechanism that allows you to use MATLAB code to perform customizations of the standard Embedded MATLAB Coder Real-Time Workshop dialog box. The Embedded MATLAB Coder software reads the `rtwTargetInfo.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Embedded MATLAB Coder session.

Using the `sl_customization` API to Register a TFL with Simulink Software

To register a TFL, you create an instance of `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function accepts one argument: a handle to a customization manager object. The function is declared as follows:

```
function sl_customization(cm)
```

The body of the `sl_customization` function invokes the `registerTargetInfo(tf1)` method to register one or more TFLs with the Simulink software. Typically, the `registerTargetInfo` function call references a local function that defines the TFLs to be registered. For example:

```
% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION
```

Below the `sl_customization` function, the referenced local function describes one or more TFLs to be registered. For example, you can declare the local function as follows:

```
% Local function to define a TFL
function thisTfl = locTflRegFcn
```

In the local function body, for each TFL to be registered, you instantiate a TFL registry entry using `tf1 = RTW.TflRegistry`. For example,

```
thisTfl = RTW.TflRegistry;
```

Then, you define the TFL properties shown in the following table within the registry entry.

TFL Property	Description
Name	String specifying the name of the TFL, as it should be displayed in the Target function library drop-down list on the Interface pane of the Configuration Parameters dialog box.
Description	String specifying a text description of the TFL, as it should be displayed in the tool tip for the TFL in the Configuration Parameters dialog box.
TableList	<p>Cell array of strings specifying the tables that make up the TFL, in descending priority order. Tables can be specified in any of the following ways:</p> <ul style="list-style-type: none"> • Name of a TFL table file on the MATLAB search path • Absolute path to a table file • Path to a table file relative to \$(MATLAB_ROOT) <p>See “Registering Multiple TFLs” on page 29-156 for examples of each type of table specification.</p>
BaseTfl	<p>String specifying the name of the TFL on which this TFL is based.</p> <hr/> <p>Note To ensure that functions, macros, and constants used by built-in blocks are available in your TFL, and to help ensure compatibility between releases, you must specify one of the default MathWorks libraries as the base TFL: 'C89/C90 (ANSI)', 'C99 (ISO)', 'GNU99 (GNU)', 'C++ (ISO)', or an equivalent alias.</p> <hr/>

TFL Property	Description
TargetHWDeviceType	Always specify {'*'}
LanguageConstraint	Cell array of strings specifying language constraint keywords. You must specify {'C++'} if your TFL includes C++ function entries or a mix of C and C++ function entries. Otherwise you can omit the field or specify it as empty.

For example:

```

thisTfl.Name = 'Sine Function Example';
thisTfl.Description = 'Demonstration of sine function replacement';
thisTfl.TableList = {'tfl_table_sinfcn'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN

```

Combining the elements described in this section, the complete `sl_customization` function for the 'Sine Function Example' TFL would appear as follows:

```

function sl_customization(cm)
% sl_customization function to register a target function library (TFL)
% for use with Simulink

% Register the TFL defined in local function locTflRegFcn
cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION

% Local function to define a TFL containing tfl_table_sinfcn
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties

```

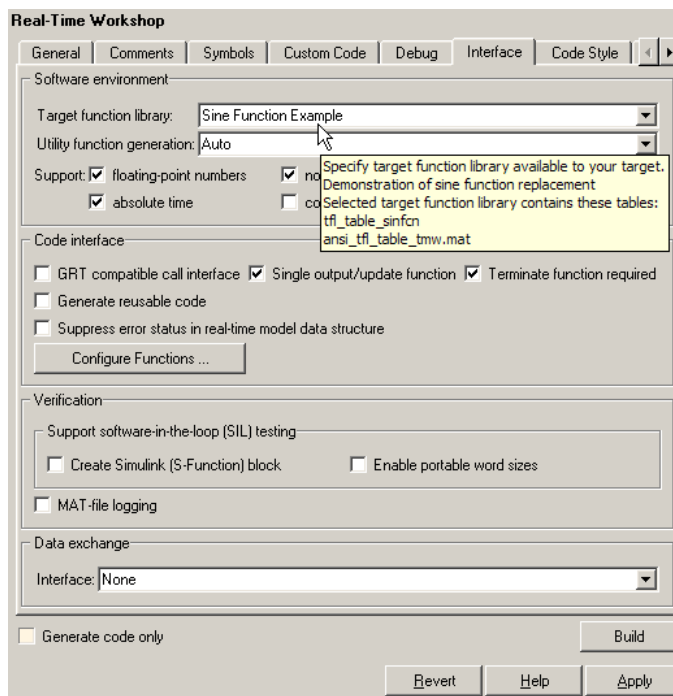
```

thisTfl.Name = 'Sine Function Example';
thisTfl.Description = 'Demonstration of sine function replacement';
thisTfl.TableList = {'tfl_table_sinfcn'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN

```

If you place the `s1_customization.m` file containing this function in the MATLAB search path or in the current working folder, the TFL is registered at each Simulink startup. The Simulink software will display the TFL in the **Target function library** drop-down list on the **Interface** pane of the Configuration Parameters dialog box. For example, the following figure shows the Configuration Parameters dialog box display, including tool tip, for the 'Sine Function Example' TFL.



Tip

- To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`.
 - To list all `sl_customization` files in the current search path, use the command `which sl_customization -all`.
 - If you disable a TFL registration (for example, by renaming the registration file `sl_customization.m` and then issuing `sl_refresh_customizations`), you may want to reset and save the **Target function library** option setting in any saved models that selected the disabled TFL.
-

Using the `rtwTargetInfo` API to Register a TFL with Embedded MATLAB Coder Software

To register a TFL for use with Embedded MATLAB Coder software, you create an instance of `rtwTargetInfo.m` and include it on the MATLAB path of the Embedded MATLAB Coder installation that you want to customize. The `rtwTargetInfo` function accepts one argument: a handle to a target registration object. The function is declared as follows:

```
function rtwTargetInfo(tr)
```

The body of the `rtwTargetInfo` function invokes the `registerTargetInfo(tfl)` method provided by the target registry object to register one or more TFLs with the Embedded MATLAB Coder software. Typically, the `registerTargetInfo` function call references a local function that defines the TFLs to be registered. For example:

```
% Register the TFL defined in local function locTflRegFcn
tr.registerTargetInfo(@locTflRegFcn);

end % End of RTWTARGETINFO
```

Below the `rtwTargetInfo` function, the referenced local function describes one or more TFLs to be registered. The format exactly matches the TFL description format previously described for Simulink use. For example, here is the Embedded MATLAB Coder equivalent of the complete TFL registration

file displayed in “Using the sl_customization API to Register a TFL with Simulink Software” on page 29-150.

```
function rtwTargetInfo(tr)
% rtwTargetInfo function to register a target function library (TFL)
% for use with emlc

% Register the TFL defined in local function locTflRegFcn
tr.registerTargetInfo(@locTflRegFcn);

end % End of RTWTARGETINFO

% Local function to define a TFL containing tfl_table_sinfcn
function thisTfl = locTflRegFcn

% Instantiate a TFL registry entry
thisTfl = RTW.TflRegistry;

% Define the TFL properties
thisTfl.Name = 'Sine Function Example';
thisTfl.Description = 'Demonstration of sine function replacement';
thisTfl.TableList = {'tfl_table_sinfcn'};
thisTfl.BaseTfl = 'C89/C90 (ANSI)';
thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

If you place the `rtwTargetInfo.m` file containing this function in the MATLAB search path or in the current working folder, the TFL is registered at each Embedded MATLAB Coder startup. The Embedded MATLAB Coder software will display the TFL in the **Target function library** drop-down list on the **Interface** pane of the Real-Time Workshop dialog box.

Tip To refresh Embedded MATLAB Coder TFL registration information within the current MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`

Registering Multiple TFLs

For an example of a TFL registration file that registers multiple TFLs, see the `sl_customization.m` file used in the TFL demo, `rtwdemo_tfl_script`. The following example illustrates the general approach, which applies equally to Simulink and Embedded MATLAB Coder TFL registration files. In this example, the three TFL tables referenced in the `TableList` fields reside at different locations, either on the MATLAB search path or at locations specified using path strings.

```
function sl_customization(cm)

    cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION

% Local function(s)
function thisTfl = locTflRegFcn
    % Register a Target Function Library for use with model: rtwdemo_tfladdsub.mdl
    thisTfl(1) = RTW.TflRegistry;
    thisTfl(1).Name = 'Addition & Subtraction Examples';
    thisTfl(1).Description = 'Demonstration of addition/subtraction op replacement';
    thisTfl(1).TableList = {'tfl_table_addsub'};
    thisTfl(1).BaseTfl = 'C89/C90 (ANSI)';
    thisTfl(1).TargetHWDeviceType = {'*'};

    % Register a Target Function Library for use with model: rtwdemo_tflmuldiv.mdl
    thisTfl(2) = RTW.TflRegistry;
    thisTfl(2).Name = 'Multiplication & Division Examples';
    thisTfl(2).Description = 'Demonstration of mult/div op repl for built-in integers';
    thisTfl(2).TableList = {'c:/work_tfl/tfl_table_muldiv'};
    thisTfl(2).BaseTfl = 'C89/C90 (ANSI)';
    thisTfl(2).TargetHWDeviceType = {'*'};

    % Register a Target Function Library for use with model: rtwdemo_tflfixpt.mdl
    thisTfl(3) = RTW.TflRegistry;
    thisTfl(3).Name = 'Fixed-Point Examples';
    thisTfl(3).Description = 'Demonstration of fixed-point operator replacement';
    thisTfl(3).TableList = ...
    {fullfile('$(MATLAB_ROOT)', 'toolbox', 'rtw', 'rtwdemos', 'tfl_demo', 'tfl_table_fixpt')};
    thisTfl(3).BaseTfl = 'C89/C90 (ANSI)';
```



```
    thisTf1(3).TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

Target Function Library Limitations

- Target function library (TFL) replacement may behave differently than you expect in some cases. For example, data types that you observe in a model do not necessarily match what the code generator determines to use as intermediate data types in an operation. To verify whether you are obtaining the function or operator replacement that you expect, inspect the generated code.
- To analyze replacement behavior, in addition to referencing the generated code and examining your TFL tables in the TFL Viewer, view the TFL cache hits and misses logged during the most recent code generation session. This approach provides information on what data types should be registered in order to achieve the desired replacement. For more information on analyzing TFL table entries, see “Examining and Validating Function Replacement Tables” on page 29-140.
- You must register a TFL in either an `sl_customization.m` file or an `rtwTargetInfo` file, but not in both files.

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

- Chapter 30, “Configuring the Target Hardware Environment”
- Chapter 31, “Interfacing With Hardware That is Not Running an Operating System (Bare Board)”
- Chapter 32, “Wind River Systems VxWorks Example Main Program”

Configuring the Target Hardware Environment

- “Configuring Support for Numeric Data” on page 30-2
- “Configuring Support for Time Values” on page 30-3
- “Setting Up Support for Non-Inlined S-Functions” on page 30-4
- “Configuring Model Function Generation and Argument Passing” on page 30-5
- “Setting Up Support for Code Reuse” on page 30-7
- “Configuring Target Function Libraries” on page 30-8

Configuring Support for Numeric Data

By default, ERT targets support code generation for integer, floating-point, nonfinite, and complex numbers.

To Generate Code that Supports...	Do...
Integer data only	Deselect Support floating-point numbers . If any noninteger data or expressions are encountered during code generation, an error message reports the offending blocks and parameters.
Floating-point data	Select Support floating-point numbers .
Nonfinite values (for example, NaN, Inf)	Select Support floating-point numbers and Support non-finite numbers .
Complex data	Select Support complex numbers .

For more information, see “Real-Time Workshop Pane: Interface” in the Real-Time Workshop reference documentation.

Configuring Support for Time Values

Certain blocks require the value of absolute time (that is, the time from the start of program execution to the present time) , elapsed time (for example, the time elapsed between two trigger events), or continuous time. Depending on the blocks used, you might need to adjust the configuration settings for supported time values.

To...	Select...
Generate code that creates and maintains integer counters for blocks that use absolute or elapsed time values (default)	Support absolute time. For further information on the allocation and operation of absolute and elapsed timers, see the “Using Timers” chapter of the Real-Time Workshop documentation. If you do not select this parameter and the model includes block that use absolute or elapsed time values, the build process generates an error.
Generate code for blocks that rely on continuous time	Support continuous time. If you do not select this parameter and the model includes continuous-time blocks, the build process generates an error.

For more information, see “Real-Time Workshop Pane: Interface” in the Real-Time Workshop reference documentation.

Setting Up Support for Non-Inlined S-Functions

To generate code for noninlined S-Functions in a model, select **Support noninlined S-functions**. The generation of noninlined S-functions requires floating-point and nonfinite numbers. Thus, when you select **Support non-inlined S-functions**, the ERT target automatically selects **Support floating-point numbers** and **Support non-finite numbers**.

When you select **Support non-finite numbers**, the build process generates an error if the model includes a C MEX S-function that does not have a corresponding TLC implementation (for inlining code generation).

Note that inlining S-functions is highly advantageous in production code generation, for example in implementing device drivers. To enforce the use of inlined S-functions for code generation, deselect **Support non-inlined S-functions**.

For more information, see “Real-Time Workshop Pane: Interface” in the Real-Time Workshop reference documentation.

Configuring Model Function Generation and Argument Passing

For ERT targets, you can configure how a model's functions are generated and how arguments are passed to the functions.

To...	Do...
Generate model function calls that are compatible with the main program module of the GRT target (<code>grt_main.c</code> or <code>.cpp</code>)	Select GRT compatible call interface and MAT-file logging . In addition, deselect Suppress error status in real-time model data structure . GRT compatible call interface provides a quick way to use ERT target features with a GRT-based custom target by generating wrapper function calls that interface to the ERT target's Embedded-C formatted code.
Reduce overhead and use more local variables by combining the output and update functions in a single <code>model_step</code> function	Select Single output/update function . Errors or unexpected behavior can occur if a Model block is part of a cycle and "Single output/update function" is enabled (the default). See "Model Blocks and Direct Feedthrough" for details.
Generate a <code>model_terminate</code> function for a model not designed to run indefinitely	Select Terminate function required . For more information, see the description of <code>model_terminate</code> .
Generate reusable, reentrant code from a model or subsystem	Select Generate reusable code . See "Setting Up Support for Code Reuse" on page 30-7 for details.
Statically allocate model data structures and access them directly in the model code	Deselect Generate reusable code . The generated code is not reusable or reentrant. See "Model Entry Points" on page 31-14 for information on the calling interface generated for model functions in this case.

To...	Do...
<p>Suppress the generation of an error status field in the real-time model data structure, <code>rtModel</code>, for example, if you do not need to log or monitor error messages</p>	<p>Select Suppress error status in real-time model data structure. Selecting this parameter can also cause the <code>rtModel</code> structure to be omitted completely from the generated code.</p> <p>When generating code for multiple integrated models, set this parameter the same for all of the models. Otherwise, the integrated application might exhibit unexpected behavior. For example, if you select the option in one model but not in another, the error status might not be registered by the integrated application.</p> <p>Do not select this parameter if you select the MAT-file logging option. The two options are incompatible.</p>
<p>Launch the Model Step Functions dialog box (see “Configuring Model Function Prototypes” on page 27-4) preview and modify the model’s <code>model_step</code> function prototype</p>	<p>Click Configure Step Function. Based on the Function specification value you select for your <code>model_step</code> function (supported values include <code>Default model-step function</code> and <code>Model specific C prototype</code>), you can preview and modify the function prototype. Once you validate and apply your changes, you can generate code based on your function prototype modifications. For more information about using the Configure Step Function button and the Model Step Functions dialog box, see Chapter 27, “Controlling Generation of Function Prototypes”.</p>

For more information, see “Real-Time Workshop Pane: Interface” in the Real-Time Workshop reference documentation.

Setting Up Support for Code Reuse

For ERT targets, you can configure how a model reuses code using the **Generate reusable code** parameter.

Pass root-level I/O as provides options that control how model inputs and outputs at the root level of the model are passed to the *model_step* function.

To...	Select...
Pass each root-level model input and output argument to the <i>model_step</i> function individually (the default)	Generate reusable code and Pass root-level I/O as > Individual arguments.
Pack root-level input arguments and root-level output arguments into separate structures that are then passed to the <i>model_step</i> function	Generate reusable code and Pass root-level I/O as > Structure reference

In some cases, selecting **Generate reusable code** can generate code that compiles but is not reentrant. For example, if any signal, *DWork* structure, or parameter data has a storage class other than *Auto*, global data structures are generated. To handle such cases, use the **Reusable code error diagnostic** parameter to choose the severity levels for diagnostics

In some cases, the Real-Time Workshop Embedded Coder software is unable to generate valid and compilable code. For example, if the model contains any of the following, the code generated would be invalid.

- An S-function that is not code-reuse compliant
- A subsystem triggered by a wide function call trigger

In these cases, the build terminates after reporting the problem.

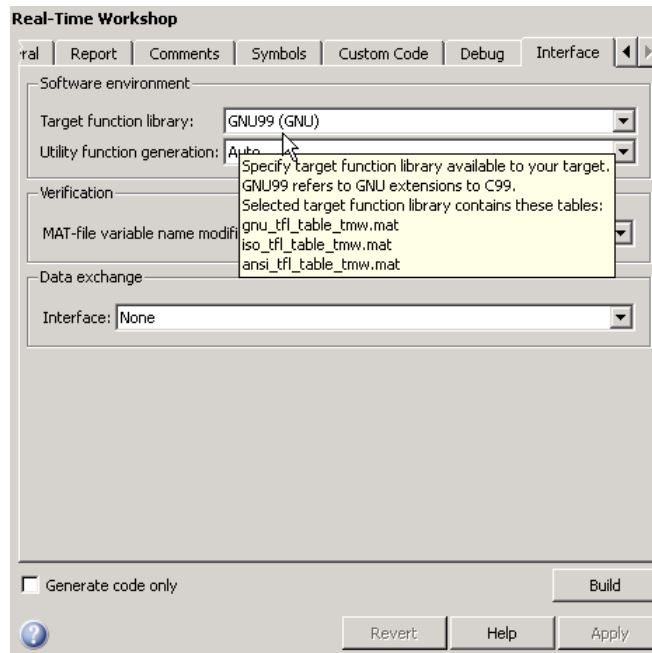
For more information, see “Real-Time Workshop Pane: Interface” in the Real-Time Workshop reference documentation.

Configuring Target Function Libraries

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 “Selecting and Viewing Target Function Libraries”. The Real-Time Workshop Embedded Coder product allows you to additionally create and register the function replacement tables that make up a TFL, as described in Chapter 29, “Replacing Math Functions and Operators Using Target Function Libraries”.

Interfacing With Hardware That is Not Running an Operating System (Bare Board)

- “About Standalone Program Execution” on page 31-2
- “Generating a Standalone Program” on page 31-3
- “Standalone Program Components” on page 31-4
- “Main Program” on page 31-5
- “rt_OneStep and Scheduling Considerations” on page 31-7
- “Model Entry Points” on page 31-14
- “Static Main Program Module” on page 31-16
- “Rate Grouping Compliance and Compatibility Issues” on page 31-21

About Standalone Program Execution

By default, the Real-Time Workshop Embedded Coder software generates *standalone* programs that do not require an external real-time executive or operating system. A standalone program requires minimal modification to be adapted to the target hardware. The standalone program architecture supports execution of models with either single or multiple sample rates.

Generating a Standalone Program

To generate a standalone program:

- 1** In the **Custom templates** subpane of the **Real-Time Workshop/Templates** pane of the Configuration Parameters dialog box, select the **Generate an example main program** option (is on by default). This enables the **Target operating system** menu.
- 2** From the **Target operating system** menu, select **BareBoardExample** (the default selection).
- 3** Generate the code.

The Real-Time Workshop Embedded Coder software generates significantly different code for multirate models depending on the following factors:

- Whether the model executes in single-tasking or multitasking mode.
- Whether or not reusable code is being generated.

These factors affect the scheduling algorithms used in generated code, and in some cases affect the API for the model entry point functions. The following sections discuss these variants.

Standalone Program Components

The core of a standalone program is the main loop. On each iteration, the main loop executes a background or null task and checks for a termination condition.

The main loop is periodically interrupted by a timer. The Real-Time Workshop function `rt_OneStep` is either installed as a timer interrupt service routine (ISR), or called from a timer ISR at each clock step.

The execution driver, `rt_OneStep`, sequences calls to the `model_step` functions. The operation of `rt_OneStep` differs depending on whether the generating model is single-rate or multirate. In a single-rate model, `rt_OneStep` simply calls the `model_step` function. In a multirate model, `rt_OneStep` prioritizes and schedules execution of blocks according to the rates at which they run.

Main Program

In this section...

“Overview of Operation” on page 31-5

“Guidelines for Modifying the Main Program” on page 31-5

Overview of Operation

The following pseudocode shows the execution of a Real-Time Workshop Embedded Coder main program.

```
main()
{
    Initialization (including installation of rt_OneStep as an
        interrupt service routine for a real-time clock)
    Initialize and start timer hardware
    Enable interrupts
    While(not Error) and (time < final time)
        Background task
    EndWhile
    Disable interrupts (Disable rt_OneStep from executing)
    Complete any background tasks
    Shutdown
}
```

The pseudocode is a design for a harness program to drive your model. The `ert_main.c` or `.cpp` program only partially implements this design. You must modify it according to your specifications.

Guidelines for Modifying the Main Program

This section describes the minimal modifications you should make in your production version of `ert_main.c` or `.cpp` to implement your harness program.

- 1 Call `model_initialize`.
- 2 Initialize target-specific data structures and hardware, such as ADCs or DACs.

- 3 Install `rt_OneStep` as a timer ISR.
- 4 Initialize timer hardware.
- 5 Enable timer interrupts and start the timer.

Note `rtModel` is not in a valid state until `model_initialize` has been called. Servicing of timer interrupts should not begin until `model_initialize` has been called.

- 6 Optionally, insert background task calls in the main loop.
- 7 On termination of the main loop (if applicable):
 - Disable timer interrupts.
 - Perform target-specific cleanup such as zeroing DACs.
 - Detect and handle errors. Note that even if your program is designed to run indefinitely, you may need to handle severe error conditions, such as timer interrupt overruns.

You can use the macros `rtmGetErrorStatus` and `rtmSetErrorStatus` to detect and signal errors.

rt_OneStep and Scheduling Considerations

In this section...
“Overview of Operation” on page 31-7
“Single-Rate Single-tasking Operation” on page 31-8
“Multirate Multitasking Operation” on page 31-9
“Multirate Single-Tasking Operation” on page 31-12
“Guidelines for Modifying rt_OneStep” on page 31-12

Overview of Operation

The operation of rt_OneStep depends upon

- Whether your model is single-rate or multirate. In a single-rate model, the sample times of all blocks in the model, and the model’s fixed step size, are the same. Any model in which the sample times and step size do not meet these conditions is termed multirate.
- Your model’s solver mode (SingleTasking versus MultiTasking)

Permitted Solver Modes for Real-Time Workshop® Embedded Coder™ Targeted Models on page 31-7 summarizes the permitted solver modes for single-rate and multirate models. Note that for a single-rate model, only SingleTasking solver mode is allowed.

Permitted Solver Modes for Real-Time Workshop Embedded Coder Targeted Models

Mode	Single-Rate	Multirate
SingleTasking	Allowed	Allowed
MultiTasking	Disallowed	Allowed
Auto	Allowed (defaults to SingleTasking)	Allowed (defaults to MultiTasking)

The generated code for `rt_OneStep` (and associated timing data structures and support functions) is tailored to the number of rates in the model and to the solver mode. The following sections discuss each possible case.

Single-Rate Single-tasking Operation

The only valid solver mode for a single-rate model is `SingleTasking`. Such models run in “single-rate” operation.

The following pseudocode shows the design of `rt_OneStep` in a single-rate program.

```
rt_OneStep()
{
    Check for interrupt overflow or other error
    Enable "rt_OneStep" (timer) interrupt
    Model_Step() -- Time step combines output, logging, update
}
```

For the single-rate case, the generated `model_step` function is

```
void model_step(void)
```

Single-rate `rt_OneStep` is designed to execute `model_step` within a single clock period. To enforce this timing constraint, `rt_OneStep` maintains and checks a timer overrun flag. On entry, timer interrupts are disabled until the overrun flag and other error conditions have been checked. If the overrun flag is clear, `rt_OneStep` sets the flag, and proceeds with timer interrupts enabled.

The overrun flag is cleared only upon successful return from `model_step`. Therefore, if `rt_OneStep` is reinterrupted before completing `model_step`, the re-interruption is detected through the overrun flag.

Re-interruption of `rt_OneStep` by the timer is an error condition. If this condition is detected `rt_OneStep` signals an error and returns immediately. (Note that you can change this behavior if you want to handle the condition differently.)

Note that the design of `rt_OneStep` assumes that interrupts are disabled before `rt_OneStep` is called. `rt_OneStep` should be noninterruptible until the interrupt overflow flag has been checked.

Multirate Multitasking Operation

In a multirate multitasking system, the Real-Time Workshop Embedded Coder software uses a prioritized, preemptive multitasking scheme to execute the different sample rates in your model.

The following pseudocode shows the design of `rt_OneStep` in a multirate multitasking program.

```

rt_OneStep()
{
    Check for base-rate interrupt overrun
    Enable "rt_OneStep" interrupt
    Determine which rates need to run this time step

    Model_Step0()      -- run base-rate time step code

    For N=1:NumTasks-1  -- iterate over sub-rate tasks
        If (sub-rate task N is scheduled)
            Check for sub-rate interrupt overrun
            Model_StepN()  -- run sub-rate time step code
        EndIf
    EndFor
}

```

Task Identifiers

The execution of blocks having different sample rates is broken into tasks. Each block that executes at a given sample rate is assigned a *task identifier* (`tid`), which associates it with a task that executes at that rate. Where there are `NumTasks` tasks in the system, the range of task identifiers is `0..NumTasks-1`.

Prioritization of Base-Rate and Subrate Tasks

Tasks are prioritized, in descending order, by rate. The *base-rate* task is the task that runs at the fastest rate in the system (the hardware clock rate). The base-rate task has highest priority (`tid 0`). The next fastest task (`tid 1`) has the next highest priority, and so on down to the slowest, lowest priority task (`tid NumTasks-1`).

The slower tasks, running at submultiples of the base rate, are called *subrate* tasks.

Rate Grouping and Rate-Specific `model_step` Functions

In a single-rate model, all block output computations are performed within a single function, `model_step`. For multirate, multitasking models, the Real-Time Workshop Embedded Coder software uses a different strategy (whenever possible). This strategy is called *rate grouping*. Rate grouping generates separate `model_step` functions for the base rate task and each subrate task in the model. The function naming convention for these functions is

```
model_stepN
```

where *N* is a task identifier. For example, for a model named `my_model` that has three rates, the following functions are generated:

```
void my_model_step0 (void);  
void my_model_step1 (void);  
void my_model_step2 (void);
```

Each `model_stepN` function executes all blocks sharing `tid N`; in other words, all block code that executes within task *N* is grouped into the associated `model_stepN` function.

Scheduling `model_stepN` Execution

On each clock tick, `rt_OneStep` and `model_step0` maintain scheduling counters and *event flags* for each subrate task. The counters are implemented in the `Timing.TaskCounters.TIDn` fields of `rtModel`. The event flags are implemented as arrays, indexed on `tid`.

The scheduling counters are maintained by the `rate_monotonic_scheduler` function, which is called by `model_step0` (that is, in the base-rate task). The function updates flags—an active task flag for each subrate and rate transition flags for tasks that exchange data—and assumes the use of a rate monotonic scheduler. The scheduling counters are, in effect, clock rate dividers that count up the sample period associated with each subrate task.

The event flags indicate whether or not a given task is scheduled for execution. `rt_OneStep` maintains the event flags based on a task counter that is maintained by code in the model's example main program (`ert_main.c`). When a counter indicates that a task's sample period has elapsed, the example main code sets the event flag for that task.

On each invocation, `rt_OneStep` updates its scheduling data structures and steps the base-rate task (`rt_OneStep` always calls `model_step0` because the base-rate task must execute on every clock step). Then, `rt_OneStep` iterates over the scheduling flags in `tid` order, unconditionally calling `model_stepN` for any task whose flag is set. This ensures that tasks are executed in order of priority.

Preemption

Note that the design of `rt_OneStep` assumes that interrupts are disabled before `rt_OneStep` is called. `rt_OneStep` should be noninterruptible until the base-rate interrupt overflow flag has been checked (see pseudocode above).

The event flag array and loop variables used by `rt_OneStep` are stored as local (stack) variables. This ensures that `rt_OneStep` is reentrant. If `rt_OneStep` is reinterrupted, higher priority tasks preempt lower priority tasks. Upon return from interrupt, lower priority tasks resume in the previously scheduled order.

Overrun Detection

Multirate `rt_OneStep` also maintains an array of timer overrun flags. `rt_OneStep` detects timer overrun, per task, by the same logic as single-rate `rt_OneStep`.

Note If you have developed multirate S-functions, or if you use a customized static main program module, see “Rate Grouping Compliance and Compatibility Issues” on page 31-21 for information about how to adapt your code for rate grouping compatibility. This adaptation lets your multirate, multitasking models generate more efficient code.

Multirate Single-Tasking Operation

In a multirate single-tasking program, by definition, all sample times in the model must be an integer multiple of the model's fixed-step size.

In a multirate single-tasking program, blocks execute at different rates, but under the same task identifier. The operation of `rt_OneStep`, in this case, is a simplified version of multirate multitasking operation. Rate grouping is not used. The only task is the base-rate task. Therefore, only one `model_step` function is generated:

```
void model_step(int_T tid)
```

On each clock tick, `rt_OneStep` checks the overrun flag and calls `model_step`, passing in `tid 0`. The scheduling function for a multirate single-tasking program is `rate_scheduler` (rather than `rate_monotonic_scheduler`). The scheduler maintains scheduling counters on each clock tick. There is one counter for each sample rate in the model. The counters are implemented in an array (indexed on `tid`) within the `Timing` structure within `rtModel`.

The counters are, in effect, clock rate dividers that count up the sample period associated with each subrate task. When a counter indicates that a sample period for a given rate has elapsed, `rate_scheduler` clears the counter. This condition indicates that all blocks running at that rate should execute on the next call to `model_step`, which is responsible for checking the counters.

Guidelines for Modifying `rt_OneStep`

`rt_OneStep` does not require extensive modification. The only required modification is to reenable interrupts after the overrun flags and error conditions have been checked. If applicable, you should also

- Save and restore your FPU context on entry and exit to `rt_OneStep`.
- Set model inputs associated with the base rate before calling `model_step0`.
- Get model outputs associated with the base rate after calling `model_step0`.

Note If you modify `rt_OneStep` to read a value from a continuous output port after each base-rate model step, see the relevant cautionary guideline below.

- In a multirate, multitasking model, set model inputs associated with subrates before calling `model_stepN` in the subrate loop.
- In a multirate, multitasking model, get model outputs associated with subrates after calling `model_stepN` in the subrate loop.

Comments in `rt_OneStep` indicate the appropriate place to add your code.

In multirate `rt_OneStep`, you can improve performance by unrolling `for` and `while` loops.

In addition, you may choose to modify the overrun behavior to continue execution after error recovery is complete.

Also observe the following cautionary guidelines:

- You should not modify the way in which the counters, event flags, or other timing data structures are set in `rt_OneStep`, or in functions called from `rt_OneStep`. The `rt_OneStep` timing data structures (including `rtModel`) and logic are critical to correct operation of any Real-Time Workshop Embedded Coder program.
- If you have customized `ert_main.c` or `.cpp` to read model outputs after each base-rate model step, be aware that selecting model options **Support: continuous time** and **Single output/update function** together may cause output values read from `ert_main` for a continuous output port to differ slightly from the corresponding output values in the model's logged data. This is because, while logged data is a snapshot of output at major time steps, output read from `ert_main` after the base-rate model step potentially reflects intervening minor time steps. To eliminate the discrepancy, either separate the generated output and update functions (clear the **Single output/update function** option) or place a Zero-Order Hold block before the continuous output port.

Model Entry Points

The following functions represent entry points in the generated code for a Simulink model.

Function	Description
<code>model_initialize</code>	Initialization entry point in generated code for Simulink model
<code>model_SetEventsForThisBaseStep</code>	Set event flags for multirate, multitasking operation before calling <code>model_step</code> for Simulink model — not generated as of Version 5.1 (R2008a)
<code>model_step</code>	Step routine entry point in generated code for Simulink model
<code>model_terminate</code>	Termination entry point in generated code for Simulink model

Note that the calling interface generated for each of these functions differs significantly depending on how you set the **Generate reusable code** option (see Chapter 30, “Configuring the Target Hardware Environment”).

By default, **Generate reusable code** is off, and the model entry point functions access model data with statically allocated global data structures.

When **Generate reusable code** is on, model data structures are passed in (by reference) as arguments to the model entry point functions. For efficiency, only those data structures that are actually used in the model are passed in. Therefore when **Generate reusable code** is on, the argument lists generated for the entry point functions vary according to the requirements of the model.

The entry points are exported with `model.h`. To call the entry-point functions from your hand-written code, add an `#include model.h` directive to your code. If **Generate reusable code** is on, you must examine the generated code to determine the calling interface required for these functions.

For more information, see the reference pages for the listed functions.

Note The function reference pages document the default (**Generate reusable code** off) calling interface generated for these functions.

Static Main Program Module

In this section...

“Overview” on page 31-16

“Rate Grouping and the Static Main Program” on page 31-17

“Modifying the Static Main Program” on page 31-18

Overview

In most cases, the easiest strategy for deploying generated code is to use the **Generate an example main program option** to generate the `ert_main.c` or `.cpp` module (see “Generating a Standalone Program” on page 31-3).

However, if you turn the **Generate an example main program** option off, you can use the module `matlabroot/rtw/c/ert/ert_main.c` as a template example for developing your embedded applications. The module is not part of the generated code; it is provided as a basis for your custom modifications, and for use in simulation. If your existing applications, developed prior to this release, depend upon a static `ert_main.c`, you may need to continue using this module.

When developing applications using a static `ert_main.c`, you should copy this module to your working directory and rename it to `model_ert_main.c` before making modifications. Also, you must modify the template makefile such that the build process creates `model_ert_main.obj` (on UNIX, `model_ert_main.o`) in the build directory.

The static `ert_main.c` contains

- `rt_OneStep`, a timer interrupt service routine (ISR). `rt_OneStep` calls `model_step` to execute processing for one clock period of the model.
- A skeletal main function. As provided, `main` is useful in simulation only. You must modify `main` for real-time interrupt-driven execution.

For single-rate models, the operation of `rt_OneStep` and the main function are essentially the same in the static version of `ert_main.c` as they are in the autogenerated version described in “About Standalone Program Execution”

on page 31-2. For multirate, multitasking models, however, the static and generated code is slightly different. The next section describes this case.

Rate Grouping and the Static Main Program

Targets based on the ERT target sometimes use a static `ert_main` module and disallow use of the **Generate an example main program** option. This may be necessary because target-specific modifications have been added to the static `ert_main.c`, and these modifications would not be preserved if the main program were regenerated.

Your `ert_main` module may or may not use rate grouping compatible `model_stepN` functions. If your `ert_main` module is based on the static `ert_main.c` module, it does not use rate-specific `model_stepN` function calls. The static `ert_main.c` module uses the old-style `model_step` function, passing in a task identifier:

```
void model_step(int_T tid);
```

By default, when the **Generate an example main program** option is off, the ERT target generates a `model_step` “wrapper” for multirate, multitasking models. The purpose of the wrapper is to interface the rate-specific `model_stepN` functions to the old-style call. The wrapper code dispatches to the appropriate `model_stepN` call with a `switch` statement, as in the following example:

```
void mymodel_step(int_T tid) /* Sample time: */
{
    switch(tid) {
        case 0 :
            mymodel_step0();
            break;
        case 1 :
            mymodel_step1();
            break;
        case 2 :
            mymodel_step2();
            break;
        default :
            break;
    }
}
```

```
    }  
}
```

The following pseudocode shows how `rt_OneStep` calls `model_step` from the static main program in a multirate, multitasking model.

```
rt_OneStep()  
{  
    Check for base-rate interrupt overflow  
    Enable "rt_OneStep" interrupt  
    Determine which rates need to run this time step  
  
    ModelStep(tid=0)    --base-rate time step  
  
    For N=1:NumTasks-1 -- iterate over sub-rate tasks  
        Check for sub-rate interrupt overflow  
        If (sub-rate task N is scheduled)  
            ModelStep(tid=N)    --sub-rate time step  
        EndIf  
    EndFor  
}
```

You can use the TLC variable `RateBasedStepFcn` to specify that only the rate-based step functions are generated, without the wrapper function. If your target calls the rate grouping compatible `model_stepN` function directly, set `RateBasedStepFcn` to 1. In this case, the wrapper function is not generated.

You should set `RateBasedStepFcn` prior to the `%include "codegenentry.tlc"` statement in your system target file. Alternatively, you can set `RateBasedStepFcn` in your `target_settings.tlc` file.

Modifying the Static Main Program

As in the generated `ert_main.c`, a few modifications to the main loop and `rt_OneStep` are necessary. See “Guidelines for Modifying the Main Program” on page 31-5 and “Guidelines for Modifying `rt_OneStep`” on page 31-12.

Also, you should replace the `rt_OneStep` call in the main loop with a background task call or null statement.

Other modifications you may need to make are

- If your model has multiple rates, the generated code does not operate correctly unless:
 - The multirate scheduling code is removed. The relevant code is tagged with the keyword `REMOVE` in comments (see also the Version 3.0 comments in `ert_main.c`).
 - Use the `MODEL_SETEVENTS` macro (defined in `ert_main.c`) to set the event flags instead of accessing the flags directly. The relevant code is tagged with the keyword `REPLACE` in comments.
- If applicable, follow comments in the code regarding where to add code for reading/writing model I/O and saving/restoring FPU context.

Note If you modify `ert_main.c` to read a value from a continuous output port after each base-rate model step, see the relevant cautionary guideline in “Guidelines for Modifying `rt_OneStep`” on page 31-12.

- When the **Generate an example main program** option is off, the Real-Time Workshop Embedded Coder software generates the file `autobuild.h` to provide an interface between the main module and generated model code. If you create your own static main program module, you would normally include `autobuild.h`.

Alternatively, you can suppress generation of `autobuild.h`, and include `model.h` directly in your main module. To suppress generation of `autobuild.h`, use the following statement in your system target file:

```
%assign AutoBuildProcedure = 0
```

- If you have cleared the **Terminate function required** option, remove or comment out the following in your production version of `ert_main.c`:
 - The `#if TERMFCN...` compile-time error check
 - The call to `MODEL_TERMINATE`
- If you do *not* want to combine output and update functions, clear the **Single output/update function** option and make the following changes in your production version of `ert_main.c`:

- Replace calls to MODEL_STEP with calls to MODEL_OUTPUT and MODEL_UPDATE.
- Remove the #if ONESTEPFCN... error check.
- The static ert_main.c module does not support the **Generate Reusable Code** option. Use this option only if you are generating a main program. The following error check raises a compile-time error if **Generate Reusable Code** is used illegally.

```
#if MULTI_INSTANCE_CODE==1
```

- The static ert_main.c module does not support the **External mode** option. Use this option only if you are generating a main program. The following error check raises a compile-time error if **External mode** is used illegally.

```
#ifdef EXT_MODE
```

Rate Grouping Compliance and Compatibility Issues

In this section...

“Main Program Compatibility” on page 31-21

“Making Your S-Functions Rate Grouping Compliant” on page 31-21

Main Program Compatibility

When the **Generate an example main program** option is off, the Real-Time Workshop Embedded Coder software generates slightly different rate grouping code, for compatibility with the older static `ert_main.c` module. See “Rate Grouping and the Static Main Program” on page 31-17 for details.

Making Your S-Functions Rate Grouping Compliant

All built-in Simulink blocks, as well as all Signal Processing Blockset blocks, are compliant with the requirements for generating rate grouping code. However, user-written multirate inlined S-functions may not be rate grouping compliant. Noncompliant blocks generate less efficient code, but are otherwise compatible with rate grouping. To take full advantage of the efficiency of rate grouping, your multirate inlined S-functions must be upgraded to be fully rate grouping compliant. You should upgrade your TLC S-function implementations, as described in this section.

Use of noncompliant multirate blocks to generate rate-grouping code generates dead code. This can cause two problems:

- Reduced code efficiency.
- Warning messages issued at compile time. Such warnings are caused when dead code references temporary variables before initialization. Since the dead code never runs, this problem does not affect the run-time behavior of the generated code.

To make your S-functions rate grouping compliant, you can use the following TLC functions to generate `ModelOutputs` and `ModelUpdate` code, respectively:

```
OutputsForTID(block, system, tid)
UpdateForTID(block, system, tid)
```

The code listings below illustrate generation of output computations without rate grouping (Listing 1) and with rate grouping (Listing 2). Note the following:

- The `tid` argument is a task identifier (`0..NumTasks-1`).
- Only code guarded by the `tid` passed in to `OutputsForTID` is generated. The `if (%<LibIsSFcnSampleHit(portName)>)` test is not used in `OutputsForTID`.
- When generating rate grouping code, `OutputsForTID` and/or `UpdateForTID` is called during code generation. When generating non-rate-grouping code, `Outputs` and/or `Update` is called.
- In rate grouping compliant code, the top-level `Outputs` and/or `Update` functions call `OutputsForTID` and/or `UpdateForTID` functions for each rate (`tid`) involved in the block. The code returned by `OutputsForTID` and/or `UpdateForTID` must be guarded by the corresponding `tid` guard:

```
if (%<LibIsSFcnSampleHit(portName)>)
```

as in Listing 2.

Listing 1: Outputs Code Generation Without Rate Grouping

```
%% multirate_blk.tlc

%implements "multirate_blk" "C"

%% Function: mdlOutputs =====
%% Abstract:
%%
%% Compute the two outputs (input signal decimated by the
%% specified parameter). The decimation is handled by sample times.
%% The decimation is only performed if the block is enabled.
%% Each ports has a different rate.
%%
%% Note, the usage of the enable should really be protected such that
%% Neach task has its own enable state. In this example, the enable
%% occurs immediately which may or may not be the expected behavior.
```

```

%%
%function Outputs(block, system) Output
/* %<Type> Block: %<Name> */
%assign enable = LibBlockInputSignal(0, "", "", 0)
{
    int_T *enabled = &%<LibBlockIWork(0, "", "", 0)>;

    %if LibGetSFcnTIDType("InputPortIdx0") == "continuous"
        %% Only check the enable signal on a major time step.
        if (%<LibIsMajorTimeStep()> && ...
            %<LibIsSFcnSampleHit("InputPortIdx0")>) {
            *enabled = (%<enable> > 0.0);
        }
    %else
        if (%<LibIsSFcnSampleHit("InputPortIdx0")>) {
            *enabled = (%<enable> > 0.0);
        }
    %endif

    if (*enabled) {
        %assign signal = LibBlockInputSignal(1, "", "", 0)
        if (%<LibIsSFcnSampleHit("OutputPortIdx0")>) {
            %assign y = LibBlockOutputSignal(0, "", "", 0)
            %<y> = %<signal>;
        }
        if (%<LibIsSFcnSampleHit("OutputPortIdx1")>) {
            %assign y = LibBlockOutputSignal(1, "", "", 0)
            %<y> = %<signal>;
        }
    }
}

%endfunction
%% [EOF] sfun_multirate.tlc

```

Listing 2: Outputs Code Generation With Rate Grouping

```

%% example_multirateblk.tlc

%implements "example_multirateblk" "C"

```

```

%% Function: mdlOutputs =====
%% Abstract:
%%
%% Compute the two outputs (the input signal decimated by the
%% specified parameter). The decimation is handled by sample times.
%% The decimation is only performed if the block is enabled.
%% All ports have different sample rate.
%%
%% Note: the usage of the enable should really be protected such that
%% each task has its own enable state. In this example, the enable
%% occurs immediately which may or may not be the expected behavior.
%%
%function Outputs(block, system) Output

%assign portIdxName = ["InputPortIdx0", "OutputPortIdx0", "OutputPortIdx1"]
%assign portTID      = [%<LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx0")>, ...
                        %<LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx0")>, ...
                        %<LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx1")>]

%foreach i = 3
    %assign portName = portIdxName[i]
    %assign tid      = portTID[i]
    if (%<LibIsSFcnSampleHit(portName)>) {
        %<OutputsForTID(block,system,tid)>
    }
%endforeach

%endfunction

%function OutputsForTID(block, system, tid) Output
/* %<Type> Block: %<Name> */
%assign enable = LibBlockInputSignal(0, "", "", 0)
%assign enabled = LibBlockIWork(0, "", "", 0)
%assign signal = LibBlockInputSignal(1, "", "", 0)

%switch(tid)
    %case LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx0")

```

```
%if LibGetSFcnTIDType("InputPortIdx0") == "continuous"
    %% Only check the enable signal on a major time step.
    if (%<LibIsMajorTimeStep()>) {
        %<enabled> = (%<enable> > 0.0);
    }
%else
    %<enabled> = (%<enable> > 0.0);
%endif
%break

%case LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx0")
    if (%<enabled>) {
        %assign y = LibBlockOutputSignal(0, "", "", 0)
        %<y> = %<signal>;
    }
    %break

%case LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx1")
    if (%<enabled>) {
        %assign y = LibBlockOutputSignal(1, "", "", 0)
        %<y> = %<signal>;
    }
    %break

%default
    %% error it out

%endswitch

%endfunction

%% [EOF] sfun_multirate.tlc
```


Wind River Systems VxWorks Example Main Program

- “Introduction to the VxWorks Example Main Program” on page 32-2
- “Task Management” on page 32-3

Introduction to the VxWorks Example Main Program

The Real-Time Workshop Embedded Coder product provides a Wind River Systems VxWorks example main program as a template for the deployment of generated code in a real-time operating system (RTOS) environment. You should read the preceding sections of this chapter as a prerequisite to working with the VxWorks example main program. An understanding of the Real-Time Workshop Embedded Coder scheduling and tasking concepts and algorithms, described in “About Standalone Program Execution” on page 31-2, is essential to understanding how generated code is adapted to an RTOS.

In addition, an understanding of how tasks are managed under the VxWorks RTOS is required. See your VxWorks documentation.

To generate a VxWorks example program:

- 1** In the **Custom templates** subpane of the **Real-Time Workshop/Templates** pane of the Configuration Parameters dialog box, select the **Generate an example main program** option (this option is on by default).
- 2** When **Generate an example main program** is selected, the **Target operating system** menu is enabled. Select **VxWorksExample** from this menu.

Some modifications to the generated code are required; comments in the generated code identify the required modifications.

Task Management

In this section...

“Overview of Operation” on page 32-3

“Single-Rate Single-tasking Operation” on page 32-3

“Multirate Multitasking Operation” on page 32-4

“Multirate Single-tasking Operation” on page 32-4

Overview of Operation

In a VxWorks example program, the main program and the base rate and subrate tasks (if any) run as prioritized tasks. The logic of a VxWorks example program parallels that of a stand-alone program; the main difference lies in the fact that base rate and subrate tasks are activated by clock semaphores managed by the operating system, rather than directly by timer interrupts.

Your application code must spawn `model_main()` as an independent VxWorks task. The task priority you specify is passed in to `model_main()`.

As with a stand-alone program, the VxWorks example program architecture is tailored to the number of rates in the model and to the solver mode (see Permitted Solver Modes for Real-Time Workshop® Embedded Coder™ Targeted Models on page 31-7). The following sections discuss each possible case.

Single-Rate Single-tasking Operation

In a single-rate, single-tasking model, `model_main()` spawns a base rate task, `tBaseRate`. In this case `tBaseRate` is the functional equivalent to `rtOneStep`. The base rate task is activated by a clock semaphore provided by the VxWorks RTOS, rather than by a timer interrupt. On each activation, `tBaseRate` calls `model_step`.

Note that the clock rate granted by the VxWorks RTOS may not be the same as the rate requested by `model_main`.

Multirate Multitasking Operation

In a multirate, multitasking model, *model_main()* spawns a base rate task and subrate tasks. Task priorities are assigned by rate.

As in a stand-alone program, rate grouping code is used (where possible) for multirate, multitasking models. The base rate task calls *model_step0*, while the subrate tasks call *model_stepN*. The base rate task calls a function that updates flags—an active task flag for each subrate and rate transition flags for tasks that exchange data. This function assumes the use of a rate-monotonic scheduler.

Multirate Single-tasking Operation

In a multirate, single-tasking model, *model_main()* spawns only a base rate task, *tBaseRate*. All rates run under this task. The base rate task is activated by a clock semaphore provided by the VxWorks RTOS, rather than by a timer interrupt. On each activation, *tBaseRate* calls *model_step*.

model_step in turn calls the *rate_scheduler* utility, which maintains the scheduling counters that determine which rates should execute. *model_step* is responsible for checking the counters.

Verifying Generated Code Applications

- Chapter 33, “Tracing Generated Code to Requirements”
- Chapter 34, “Verifying Generated Code”
- Chapter 35, “Rapid Prototyping On a Target System”
- Chapter 36, “Verifying Generated Source Code With Software-in-the-Loop Simulation”
- Chapter 37, “Verifying a Component in the Target Environment”
- Chapter 38, “Verifying a Component by Building a Complete Real-Time Target Environment”
- Chapter 39, “Verifying Compiled Object Code with Processor-in-the-Loop Simulation”
- Chapter 40, “Verifying Numerical Equivalence of Results with Code Generation Verification API”

Tracing Generated Code to Requirements

- “About Generated Code and Requirements Traceability” on page 33-2
- “Goals of Generated Code and Requirements Traceability” on page 33-3

About Generated Code and Requirements Traceability

Assuming that you have captured application requirements in a document, spreadsheet, data base, or requirements management tool, using a tool such as Simulink Report Generator, Microsoft® Word, Microsoft Excel, raw HTML, Telelogic®, or DOORS®, you can use interactive traceability and traceability reports to validate whether generated code meets the documented requirements. These mechanisms provide a way to trace generated code back to documented requirements and generate traceability reports.

Goals of Generated Code and Requirements Traceability

For example, you can

- Associate requirements documents with objects in a concept model and generate a report on requirements associated with that model. For more information, see:
 - `slvndemo_fuelsys_docreq`
 - “Requirements Linking and Traceability” in the Simulink Verification and Validation documentation
 - Bidirectional tracing in Microsoft Word, Microsoft Excel, HTML, and Telelogic DOORS
- Include requirements tags in generated code. For more information, see:
 - `rtwdemo_requirements`
 - “Including Requirements Information with Generated Code” in the Simulink Verification and Validation documentation
- Trace model blocks and subsystems to generated code and vice versa. For more information, see:
 - `rtwdemo_hyperlinks`
 - “About Traceability” on page 34-2

Verifying Generated Code

- “Traceability for Production Code Generation” on page 34-2
- “Checking Code Correctness” on page 34-11

Traceability for Production Code Generation

In this section...
“About Traceability” on page 34-2
“Tracing Code to Model Objects Using Hyperlinks” on page 34-2
“Tracing Model Objects to Generated Code” on page 34-4
“Reloading Existing Traceability Information” on page 34-6
“Customizing Traceability Reports” on page 34-7
“Traceability Limitations” on page 34-9

About Traceability

The Real-Time Workshop product introduces traceability capabilities:

- “About Code Traceability”
- “Format of Traceability Tags”
- “Examples of Tagged Code”
- “Tracing Code To Blocks Using `hilite_system`”
- “Traceability Limitations”

The Real-Time Workshop Embedded Coder product includes traceability capabilities to support:

- “Tracing Code to Model Objects Using Hyperlinks” on page 34-2
- “Tracing Model Objects to Generated Code” on page 34-4
- “Reloading Existing Traceability Information” on page 34-6
- “Customizing Traceability Reports” on page 34-7

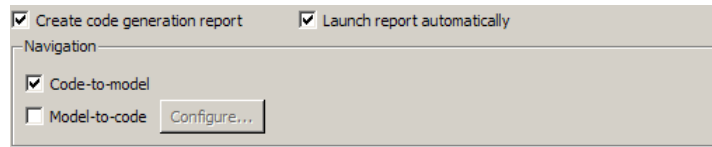
Tracing Code to Model Objects Using Hyperlinks

When using the Real-Time Workshop product, you can trace code to model objects using the `hilite_system` command. The Real-Time Workshop Embedded Coder product simplifies traceability with the use of hyperlinks

in HTML code generation reports. The reports display hyperlinks in “Regarding,” “Outport,” and other comment lines in generated code. You can highlight the corresponding block or subsystem in the model diagram by clicking the hyperlinks.

To use hyperlinks for tracing code to model objects:

- 1 Open the model and make sure it is configured for an ERT target.
- 2 In the Configuration Parameters dialog box, select **Real-Time Workshop > Report Create code generation report**. The parameter is selected by default. When selected, the parameter enables and selects **Launch report automatically** and **Code-to-model**.



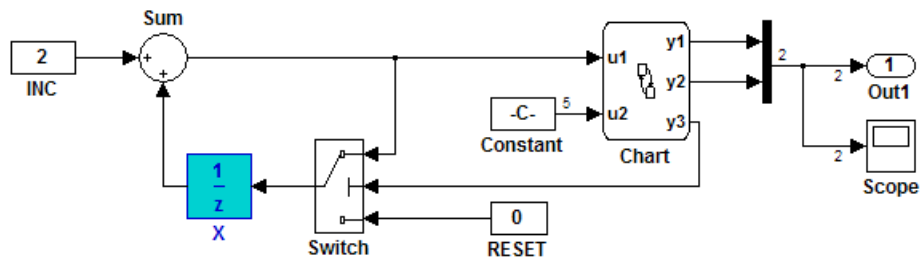
- 3 Build or generate code for the model. An HTML code generation report is displayed.
- 4 In the HTML report window, click hyperlinks to highlight source blocks. For example, generate an HTML report for model `rtwdemo_hyperlinks`. In the generated code for the model step function in `rtwdemo_hyperlinks.c`, click the first `UnitDelay` block hyperlink .

```

36  /* Model step function */
37  void rtwdemo_hyperlinks_step(void)
38  {
39  {
40      boolean_T sf_aVarTruthTableCondition;
41      boolean_T sf_aVarTruthTableCondition_0;
42      boolean_T sf_aVarTruthTableCondition_1;
43      real_T rtb_Sum;
44
45      /* Sum: '<Root>/Sum' incorporates:
46       * Constant: '<Root>/INC'
47       * UnitDelay: '<Root>/X'
48       */
49      rtb_Sum = 2.0 + rtDWork.X;
50

```

In the model window, the corresponding `UnitDelay` block is highlighted.



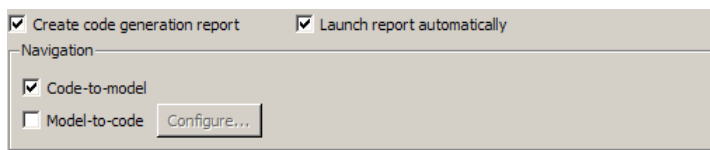
For more information on generating HTML code generation reports or using the `hilite_system` command to trace code to blocks, see the following topics in the Real-Time Workshop documentation:

- “Generating Reports for Code Reviews and Traceability Analysis”
- “Tracing Generated Code”

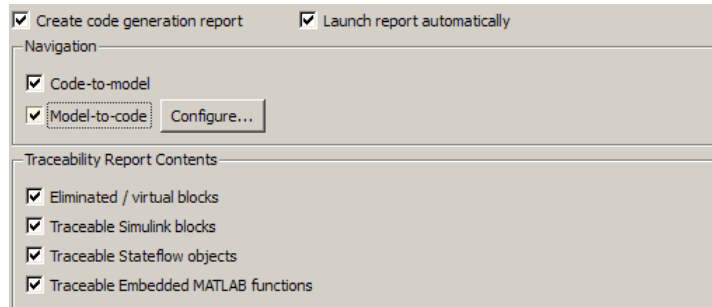
Tracing Model Objects to Generated Code

To trace model objects to generated code:

- 1 Open the model and make sure it is configured for an ERT target.
- 2 In the Configuration Parameters dialog box, select **Real-Time Workshop > Report > Create code generation report**. The parameter is selected by default. When selected, the parameter enables and selects the **Launch report automatically** and **Code-to-model** parameters.



- 3 Select **Model-to-code**.



This parameter:

- Enables the **Configure** button, which opens a dialog box for loading existing trace information.
 - Enables and selects parameters for customizing the content of a traceability report.
- 4** Build or generate code for the model. An HTML code generation report is displayed.
 - 5** In the model window, right-click a model object.
 - 6** In the context menu, select **Real-Time Workshop > Navigate to Code**. In the HTML code generation report, you see the first instance of highlighted code generated for the model object. In the left pane of the report, numbers that appear to the right of generated file names indicate the total number of highlighted lines in each file. The following figure shows the result of tracing the Unit Delay block in model `rtwdemo_hyperlinks`.

```

37 void rtwdemo_hyperlinks_step(void)
38 {
39     {
40         boolean_I sf_aVarTruthTableCondition;
41         boolean_I sf_aVarTruthTableCondition_0;
42         boolean_I sf_aVarTruthTableCondition_1;
43         real_T rtb_Sum;
44
45         /* Sum: '<Root>/Sum' incorporates:
46          * Constant: '<Root>/INC'
47          * UnitDelay: '<Root>/X'
48          */
49         rtb_Sum = 2.0 + rtDWork.X;
50
51         /* Stateflow: '<Root>/Chart' incorporates:
52          * Constant: '<Root>/Constant'
53          */
54         /* Gateway: Chart */
55         /* During: Chart */

```

To navigate through multiple instances of highlighted lines, click **Previous** and **Next**.

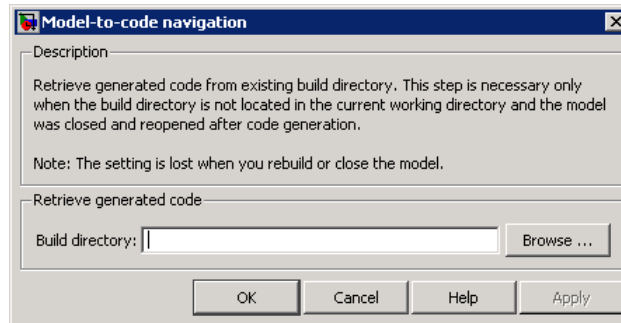
If you close and reopen a model, the **Navigate to Code** context menu option might not be available. This occurs because Real-Time Workshop Embedded Coder cannot find a build directory for your model in the current working directory. To address this, do one of the following:

- Reset the current working directory to the parent directory of the existing build directory.
- Select **Model-to-code** and rebuild the model. This regenerates the build directory into the current working directory.
- Click **Configure** and in the Model-to-code navigation dialog box, reload the existing trace information.

Reloading Existing Traceability Information

To reload existing traceability information for a model:

- 1 In the Configuration Parameters dialog box, click **Real-Time Workshop > Report > Configure**. The Model-to-code navigation dialog box opens.



- 2** In the **Build directory** field, type or browse to the build directory that contains the existing traceability information.

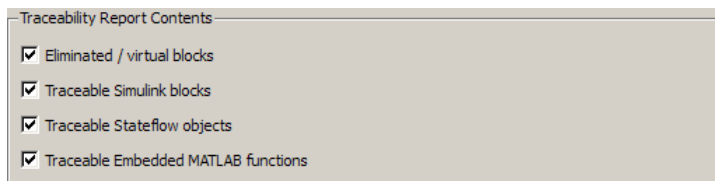
If you close and reopen a model, the **Navigate to Code** context menu option might not be available. This occurs because Real-Time Workshop Embedded Coder cannot find a build directory for your model in the current working directory. To fix this without having to reset the current working directory or rebuild the model, do the following:

- 1** Click **Configure** to open the Model-to-code navigation dialog box.
- 2** In the Model-to-code navigation dialog box, click **Browse**.
- 3** Browse to the build directory for your model, and select the directory. The build directory path is displayed in the **Build directory** field, as shown in the preceding figure.
- 4** Click **Apply** or **OK**. This loads traceability information from the earlier build into your Simulink session, provided that you selected **Model-to-code** for the build.
- 5** Right-click **Real-Time Workshop > Navigate to Code** to open the context menu and trace a model object to corresponding code.

Customizing Traceability Reports

In the Configuration Parameters dialog box, the **Real-Time Workshop > Report > Traceability Report Contents** section lists parameters you can select and clear to customize the content of your

traceability reports. By default, all parameters are selected, as shown in the following figure.



Select or clear any combination of the following:

- **Eliminated / virtual blocks** (account for blocks that are untraceable)
- **Traceable Simulink blocks**
- **Traceable Stateflow objects**
- **Traceable Embedded MATLAB functions**

If you select all parameters, you get a complete mapping between model elements and the generated code.

The following figure shows the top section of the traceability report generated by selecting all traceability content parameters for model `rtwdemo_hyperlinks`.

Traceability Report for rtwdemo_hyperlinks

Table of Contents

1. [Eliminated / Virtual Blocks](#)
2. [Traceable Simulink Blocks / Stateflow Objects / Embedded MATLAB Scripts](#)
 - o [rtwdemo_hyperlinks](#)
 - o [rtwdemo_hyperlinks/Chart](#)
 - o [rtwdemo_hyperlinks/Chart:43](#)

Eliminated / Virtual Blocks

Block Name	Comment
<Root>/Build ERT	Empty SubSystem
<Root>/Mux	Mux
<Root>/Scope	Eliminated unused block
<Root>/View RTW Report	Empty SubSystem

Traceable Simulink Blocks / Stateflow Objects / Embedded MATLAB Scripts

Root system: [rtwdemo_hyperlinks](#)

Object Name	Code Location
<Root>/Chart	rtwdemo_hyperlinks.c:16, 51, 251 rtwdemo_hyperlinks.h:34, 35, 36, 42, 43, 44, 45, 46, 47, 48, 49
<Root>/Constant	rtwdemo_hyperlinks.c:52

Traceability Limitations

In addition to the Real-Time Workshop traceability limitations, the following limitations apply to reports generated by Real-Time Workshop Embedded Coder software.

- Under the following conditions, model-to-code traceability is disabled for a block if the block name contains:
 - A single quote (').
 - An asterisk (*), that causes a name-mangling ambiguity relative to other names in the model. This name-mangling ambiguity occurs if in a block name or at the end of a block name, an asterisk precedes or follows a slash (/).

- The character `(char(255))`.
- You cannot trace blocks representing the following types of subsystems to generated code:
 - Virtual subsystems
 - Masked subsystems
 - Nonvirtual subsystems for which code has been optimized away

If you cannot trace a subsystem at subsystem level, you might be able to trace individual blocks within the subsystem.

Checking Code Correctness

In this section...

“About Checking Code Correctness” on page 34-11

“How To Check Code Correctness” on page 34-11

“PolySpace Analysis Options” on page 34-12

About Checking Code Correctness

Checking code correctness involves verifying that no compile-time, link-time, or run-time errors (for example, an overflow, divide by zero, or out-of-bounds array access) are in the source code.

How To Check Code Correctness

You can rely on integrated development environment (IDE) tools to detect and facilitate correction of compile-time and link-time errors. However, it is more difficult to detect and correct run-time errors, such as overflows and division by zero. Some methods of detecting run-time errors include

- Running the executable and analyzing the results
- Inserting code instrumentation, such as print statements
- Writing and running tests

When checking the correctness of code generated by Real-Time Workshop technology, you also have the option of using PolySpace® products. PolySpace products are based on verification technology that uses formal methods to detect and mathematically prove whether classes of run-time errors, such as overflows, exist.

In addition, the PolySpace Model Link™ SL product lets you trace the results reported by PolySpace® Client™ for C/C++ software back to your Simulink model.

For more information about using PolySpace products, see the PolySpace documentation.

PolySpace Analysis Options

This section describes recommended Analysis options for verifying code generated with Real-Time Workshop Embedded Coder software. The MathWorks recommends setting these options in your PolySpace project before verifying generated code.

If you have PolySpace Model Link SL software, you can specify the Analysis options for your PolySpace Project by selecting **Tools > PolySpace > PolySpace Utilities > Configure Project** in the Simulink model window.

Option	Recommended Value	Comments
General Options		
-author	<your name>	Specifies the name of the author of the verification
-date	<date stamp>	Specifies a date stamp for the verification in dd/mm/yyyy format. This information is labelled in the GUI as Date. You can specify alternate date formats using Edit > Preferences .
-desktop	Selected – When checking MISRA Compliance (code analysis). Not Selected – When checking runtime errors (code verification)	Specifies whether verification occurs on the PolySpace Client or PolySpace Server. The MathWorks recommends using the Client to perform MISRA analysis and the Server for code verification.
-I	See Comments	Specifies the name of a folder to include when compiling C sources. You can specify only one folder for each I, but can use the option multiple times. A script to automatically determine -I based on buildInfo is available

Option	Recommended Value	Comments
-prog	< <i>session identifier</i> >	<p>Specifies the application name. Use characters valid for Unix file names.</p> <p>This information is labelled in the GUI as Session identifier.</p> <p>For example, <code>New_Project</code>.</p>
-results-dir	< <i>results dir</i> >	<p>Specifies the directory in which PolySpace software saves verification results. You can specify a relative path. However, be careful if you plan to launch verification remotely over a network, or if you plan to copy the project configuration file using the "Save as" option.</p>
-sources -sources-list-file	See Comments	<p>Specifies a list of source files to be verified. You must enclose the list of source files in double-quotes, separated by commas.</p> <ul style="list-style-type: none"> • -sources "file1[file2[...]]" (Linux and Solaris™) - • -sources "file1[,file2[, ...]]" (Windows®, Linux and Solaris) • -sources-list-file file_name (not a graphical option)

Option	Recommended Value	Comments
		<p>You can specify multiple files using UNIX® standard wildcards.</p> <p>The software compiles the source files in the order in which you specify them.</p> <p>If you do not specify any files, the software verifies all files in the source directory in alphabetical order.</p> <hr/> <p>Note The specified files must have valid extensions: * . (c C cc cpp CPP cxx CXX)</p> <hr/> <p>A script to automatically determine -sources based on buildInfo is available.</p>
-verif-version	1.0	Specifies the version identifier of the verification. You can use this option to identify multiple verifications of the same project. This information is identified in the GUI as the Version.
Target/Compilation Options		
-d	See Comments	<p>Defines macro compiler flags used during compilation.</p>

Use one `d` for each line of the RTW generated `defines.txt` file.

PolySpace Model Link SL does not do this by default.

Option	Recommended Value	Comments
-OS-target	Visual	<p>Specifies the operating system target for PolySpace stubs.</p> <p>This information allows the verification to use appropriate system definitions during preprocessing in order to analyze the included files properly.</p>
-target	i386	<p>Specifies the target processor type. This allows the verification to consider the size of fundamental data types and the endianness of the target machine.</p> <p>You can configure and specify generic targets. For more information, see Setting Up Project for Generic Target Processors in the PolySpace Products for C User's Guide.</p>
Compliance with standards Options		
-dos	Selected	<p>You must select this option if the contents of the include or source directory comes from a DOS or Windows file system. The option allows the verification to deal with upper/lower case sensitivity and control characters issues. Concerned files are:</p> <ul style="list-style-type: none"> • Header files – All include folders specified (-I option) • Source files – All source files selected for the

Option	Recommended Value	Comments
		verification (-sources option)
-misra2	[all-rules file_name]	<p>Specifies that the software checks coding rules in conformity to MISRA-C:2004. All MISRA checks are included in the log file of the verification. Options:</p> <ul style="list-style-type: none"> • all-rules – Checks all available MISRA C® rules. Any violation of MISRA C rules is considered a warning. • filename – Specifies an ASCII file containing a list of MISRA® rules to check.
-includes-to-ignore	<MSVC dir>\VC\include	Specifies files or folders that are excluded from MISRA rules checking (all files and subfolders within the selected folder). This option is useful when you have non-MISRA C conforming include headers.
PolySpace inner settings Options		
-main-generator-writes-variables	public	<p>Specifies how the generated main initializes global variables.</p> <p>By selecting public, every variable except static and const variables are assigned a "random" value, representing the full range of possible values</p>

Option	Recommended Value	Comments
-main-generator-calls	unused	<p>Specifies how the generated main calls functions.</p> <p>By selecting unused, every function is called by the generated main unless it is called elsewhere by the code undergoing verification.</p>
-ignore-float-rounding	Selected	<p>Specifies how the verification rounds floats.</p> <p>If this option is not selected, the verification rounds floats according to the IEEE® 754 standard – simple precision on 32-bits targets and double precision on targets that define double as 64-bits.</p> <p>When you select this option, the verification performs exact computation.</p> <p>Selecting this option can lead to results that differ from "real life," depending on the actual compiler and target. Some paths may be reachable (or not reachable) for the verification while they are not reachable (or are reachable) for the actual compiler and target.</p> <p>However, this option reduces the number of unproven checks caused by float approximation.</p>
Precision/Scaling Options		

Option	Recommended Value	Comments
-0	2	<p>Specifies the precision level for the verification.</p> <p>Higher precision levels provide higher selectivity at the expense of longer verification time.</p> <p>The MathWorks recommends you begin with the lowest precision level. You can then address red errors and gray code before relaunching PolySpace verification using higher precision levels.</p> <p>Benefits:</p> <p>A higher precision level contributes to a higher selectivity rate, making results review more efficient and hence making bugs in the code easier to isolate.</p> <p>The precision level specifies the algorithms used to model the program state space during verification:</p> <ul style="list-style-type: none"> • -00 corresponds to static interval verification. • -01 corresponds to complex polyhedron model of domain values. • -02 corresponds to more complex algorithms to closely model domain values (a mixed approach with

Option	Recommended Value	Comments
		<p>integer lattices and complex polyhedrons).</p> <ul style="list-style-type: none"> -03 is suitable only for units smaller than 1,000 lines of code. For such code, selectivity may reach as high as 98%, but verification may take up to an hour per 1,000 lines of code.
-from	scratch	<p>Specifies the phase from which verification starts.</p> <p>You can use this option on an existing verification to elaborate on the results that you have already obtained. For example, if a verification ran <code>-to pass1</code>, you can restart the verification <code>-from pass1</code> to reduce verification time.</p> <p>Note the following:</p> <ul style="list-style-type: none"> This option can be used only for client verifications. All server verifications start from <code>scratch</code>. All options other than <code>scratch</code> can be used only if the previous verification was launched using the option <code>-keep-all-files</code>. You cannot use this option if you modify the source code between verifications.
-to	c-compile – When checking MISRA compliance only. pass0	Specifies the phase after which the verification stops. Each

Option	Recommended Value	Comments
	<p>– When verifying code for the first time.</p> <p>pass4 – When performing subsequent verifications of code.</p>	<p>verification phase improves the selectivity of your results, but increases the overall verification time.</p> <p>Improved selectivity can make results review more efficient, and hence make bugs in the code easier to isolate.</p> <p>The MathWorks recommends you begin by running <code>-to pass0</code> (Software Safety Analysis level 0) You can then address red errors and gray code before relaunching verification using higher integration levels.</p>

Default PolySpace Analysis Options

When using the PolySpace Model Link SL product, the software sets the following Analysis options by default:

```

-sources path_to_source_code
-desktop
-D PST_ERRNO
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-ignore-float-rounding
-OS-target no-predefined-OS
-allow-ptr-arith-on-struct
-results-dir results
    
```

Note *matlabroot* is the MATLAB installation directory.

Rapid Prototyping On a Target System

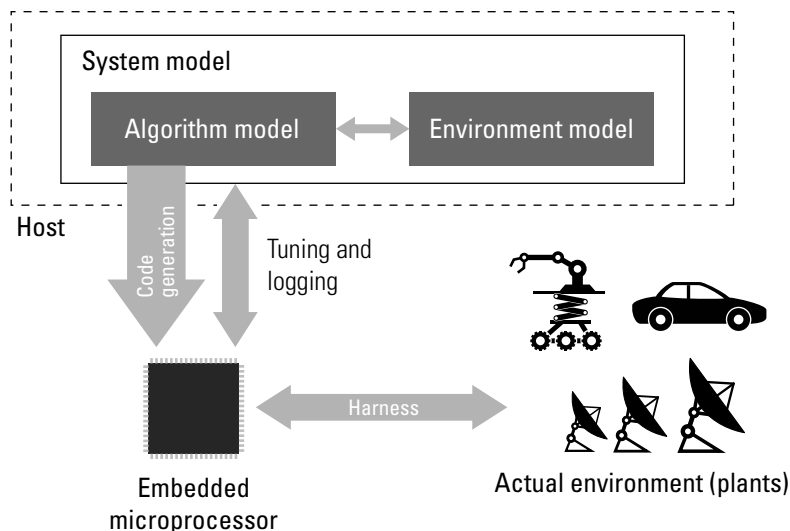
- “About On-Target Rapid Prototyping” on page 35-2
- “Goals of On-Target Rapid Prototyping” on page 35-3
- “Optimizing Generated Code for an Embedded Processor With On-Target Rapid Prototyping” on page 35-4

About On-Target Rapid Prototyping

After you refine a detailed software design, you are ready to generate code to run on an embedded microprocessor and optimize the code with on-target rapid prototyping. During on-target rapid prototyping, you run generated code in real time, tune parameters, and monitor real-time data on the same processor that you plan to use in mass production, or a close equivalent to it.

Real-Time Workshop technology provides a framework for on-target rapid prototyping. You can generate code from your model and then assess, interact with, and optimize the code using real embedded compilers and hardware. This effort can help determine whether your algorithm can fit on or run fast enough for production devices, which typically have limited processor resources.

The following figure shows an example of an on-target rapid prototyping environment.



Goals of On-Target Rapid Prototyping

Assuming that you have a detailed software design and an embedded microprocessor target, you can use on-target rapid prototyping to:

- Refine the concept model of your component or system
- Test and validate model functionality in real time
- Test hardware
- Obtain real-time profiles and code metrics for analysis and sizing based on an embedded processor
- Assess the feasibility of an algorithm based on integration with environment or plant hardware

Optimizing Generated Code for an Embedded Processor With On-Target Rapid Prototyping

To do on-target rapid prototyping:

- 1** Generate the source code for your models, integrate the code into your production build environment, and run it on existing hardware. For more information see:
 - “Testing and Refining a Model With Rapid Prototyping” in the Real-Time Workshop documentation
 - “Selecting and Configuring a Target ”
 - “Interfacing With a Real-Time Operating System ”
- 2** Integrate existing, externally written C or C++ code with your model for simulation and code generation. For more information, see “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation.
- 3** Use a third-party integrated development environment (IDE) or makefile with a MathWorks link product, third-party product, or custom integration to build an executable for the embedded microprocessor.
- 4** To monitor signals, tune parameters, and log data as the embedded microprocessor controls the actual environment or plant, use the following MathWorks products:
 - Embedded IDE Link product for integration with selected IDEs
 - Target Support Package product for deployment on selected embedded processors
- 5** If using custom integration, use a Real-Time Workshop Embedded Coder runtime interface option, such as external mode, C API, or ASAP2 file generation, to monitor and tune signals. Also consider using the Vehicle Network Toolbox™ product if you are developing a solution based on a controller-area network (CAN).

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

- “About Software-in-the-Loop Simulation” on page 36-2
- “Verifying Code Using the SIL Simulation Mode” on page 36-3
- “Choosing a SIL Approach” on page 36-5
- “Running a Complete Model as a SIL Simulation” on page 36-12
- “Running a Referenced Model as a SIL Simulation” on page 36-14
- “SIL Code Interfaces” on page 36-16
- “Configuring a SIL Simulation” on page 36-18
- “Verifying Code Using a SIL Block” on page 36-30
- “Configuring Hardware Implementation Settings” on page 36-31
- “SIL Support and Limitations” on page 36-34

About Software-in-the-Loop Simulation

The Real-Time Workshop Embedded Coder product supports software-in-the-loop (SIL) simulation. SIL simulation involves compiling and running production source code on your host computer. For more information, see:

- “Verifying Code Using the SIL Simulation Mode” on page 36-3
- “Verifying Code Using a SIL Block” on page 36-30

Verifying Code Using the SIL Simulation Mode

In this section...
“About SIL Simulation Mode” on page 36-3
“Comparison of SIL and PIL Simulation” on page 36-3

About SIL Simulation Mode

Real-Time Workshop Embedded Coder provides a Software-in-the-Loop (SIL) simulation mode that allows you to run top models or Model blocks as a SIL simulation. With this mode, you can:

- Verify generated source code without modifying the original model. See “Configuring a Top-Model SIL Simulation” on page 36-18.
- Reuse test harnesses for the original model with the generated source code. See “Configuring a Model Block SIL Simulation” on page 36-20

This mode, which executes code on the host computer, provides a convenient alternative to processor-in-the-loop (PIL) simulation when target hardware is not available. For information about how you choose a SIL simulation, see “Choosing a SIL Approach” on page 36-5.

You can also collect code coverage metrics for generated source code when you run a top-model or Model block SIL simulation. See “Using a Code Coverage Tool in a SIL Simulation” on page 36-22.

For examples of SIL verification, see `rtwdemo_sil_pil_script`. For information about how you verify that your model is correctly configured for a SIL simulation, see “Verifying a SIL or PIL Configuration” on page 36-26.

Comparison of SIL and PIL Simulation

Use SIL or PIL simulation to verify automatically generated code by comparing the results with a normal mode simulation. With SIL, you can easily verify the behavior of production source code on your host computer. However, you cannot verify exactly the same code that is subsequently compiled for your target hardware because the code must be compiled for your host computer (that is, 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 on either real target hardware or an instruction set simulator. See “About Processor-In-the-Loop Simulation” on page 39-2

You can use any of the following verification approaches:

- *Software-in-the-loop* (SIL). You simulate target behavior on the host computer. There are two approaches for SIL:
 - Select the SIL simulation mode (for top model or model-block). The generated production code is compiled and executed on the host computer as a separate process, independent of the MATLAB process. See “Verifying Code Using the SIL Simulation Mode” on page 36-3.
 - Create a SIL block. The software runs generated code through an S-function wrapper on the host computer. The SIL S-function links directly with the generated code, so that the generated code runs inside the MATLAB process. See “Verifying Code Using a SIL Block” on page 36-30.

In both SIL cases, execution is host/host and non-real-time.

- *Processor-in-the-loop* on non-host hardware. Test the generated code as cross-compiled object code on the target processor or equivalent instruction set simulator. PIL exercises the same object code used in production software. PIL execution is host/target and non-real time.

To decide which verification approach you want to use, see “Choosing a SIL Approach” on page 36-5 and “Choosing a PIL Approach” on page 39-5.

Choosing a SIL Approach

In this section...
“About Choosing a SIL Simulation” on page 36-5
“When to Use Top-Model SIL” on page 36-5
“When to Use Model Block SIL” on page 36-6
“When to Use the SIL Block” on page 36-10

About Choosing a SIL Simulation

This section describes how to choose the SIL verification approach for your needs.

The following table compares the code interfaces and API support for top-model SIL and the Model block with SIL.

SIL Feature	Standalone Code Interface	Model Reference Code Interface
Top-model SIL	Yes	No (but you can include Model blocks inside your top model)
Model block SIL mode	No	Yes
SIL block	Yes	No

See `rtwdemo_sil_pil_script` for examples comparing:

- SIL block for SIL Simulation
- SIL Simulation for Model Blocks
- SIL Simulation for Top Models

When to Use Top-Model SIL

Use top-model SIL if you want to:

- Verify code generated for a top model (standalone code interface).

- Load test vectors or stimulus inputs from the MATLAB workspace.
- Switch the entire model between normal, SIL, or PIL simulation modes.

For an example, see `rtwdemo_sil_pil_script`.

When to Use Model Block SIL

Use Model block SIL if you want to:

- Verify code generated for referenced models (model reference code interface).
- Provide a test harness model (or a system model) to generate test vector or stimulus inputs.
- Switch a Model block between normal, SIL, or PIL simulation modes.

See “Modeling Scenarios with the Model Block” on page 36-6.

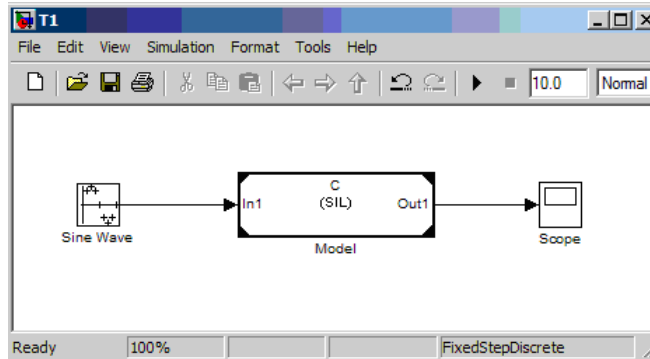
Modeling Scenarios with the Model Block

You can use the Model block to test single components or a whole hierarchy of model reference components. For example, you can select a single leaf component for SIL verification. Later in the development cycle, as your components become integrated into a larger system, you can select a hierarchy of components for SIL verification.

You must deploy your Model block component code as part of a standalone executable. The following examples show ways of testing your component.

- “Testing a Model Reference Component in SIL Mode” on page 36-6
- “Deploying Through an Atomic Subsystem” on page 36-7
- “Deploying Through a Top Model” on page 36-9

Testing a Model Reference Component in SIL Mode. You can test a model reference component or hierarchy of components by placing a Model block in a test harness model, as shown in model T1.



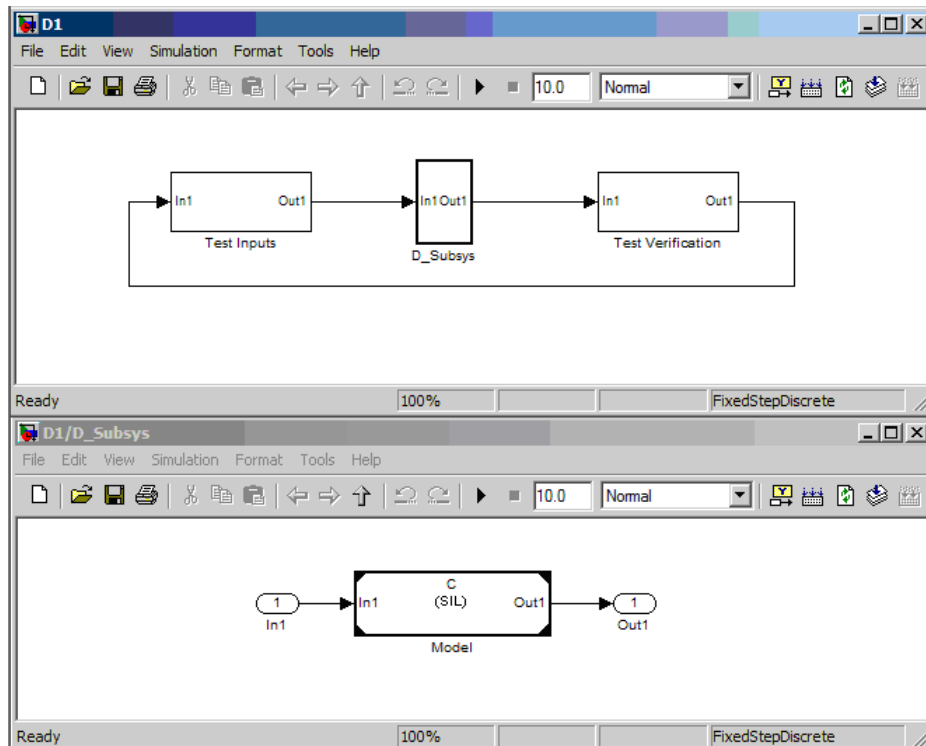
To test the component:

- 1 Set the simulation mode of component C to SIL mode.
- 2 Simulate the model to run component C in SIL mode, and test its Real-Time Workshop model reference target.

Note If the model reference target code interface for component C does not already exist, simulating the model generates it. For more information about the model reference target code interface, see “SIL Code Interfaces” on page 36-16.

The following deployment scenarios reuse the model reference target of component C. This reuse ensures that you test exactly the same object code that you deploy.

Deploying Through an Atomic Subsystem. To generate code with the standalone interface for deployment, place a Model block inside an atomic subsystem, as shown by model D1.



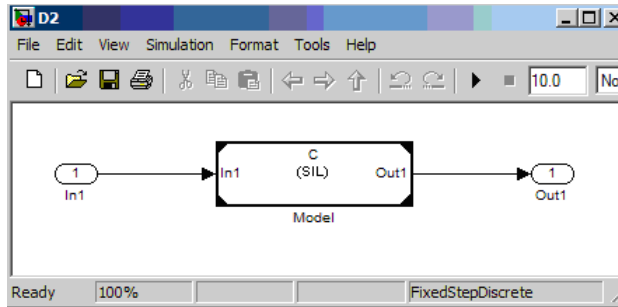
To create standalone code, perform a subsystem build of `D_Subsys`. The standalone code calls the Real-Time Workshop model reference target of component `C`.

To test the component:

- 1** Set the simulation mode of component `C` to SIL mode.
- 2** Simulate the model to run component `C` in SIL mode and test its Real-Time Workshop model reference target.

You can place multiple `Model` blocks and other blocks into the model to deploy a whole system of components.

Deploying Through a Top Model. To generate code with the standalone interface for deployment, place the Model block inside a top model, as shown by model D2.

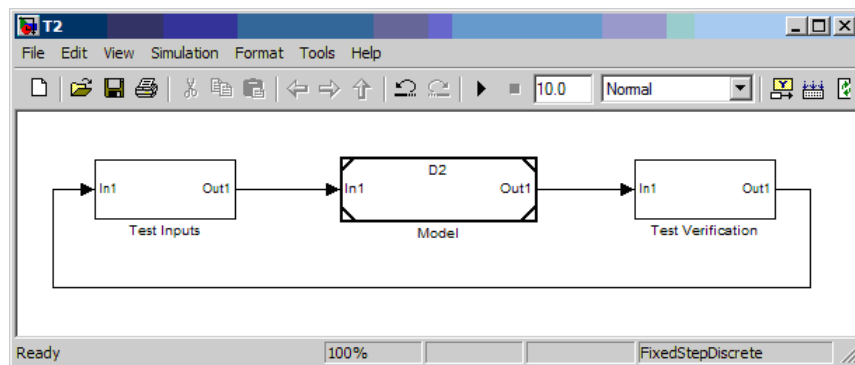


To create standalone code, perform a build of D2. The standalone code calls the Real-Time Workshop model reference target of component C.

You can place multiple Model blocks and other blocks into the model to deploy a whole system of components.

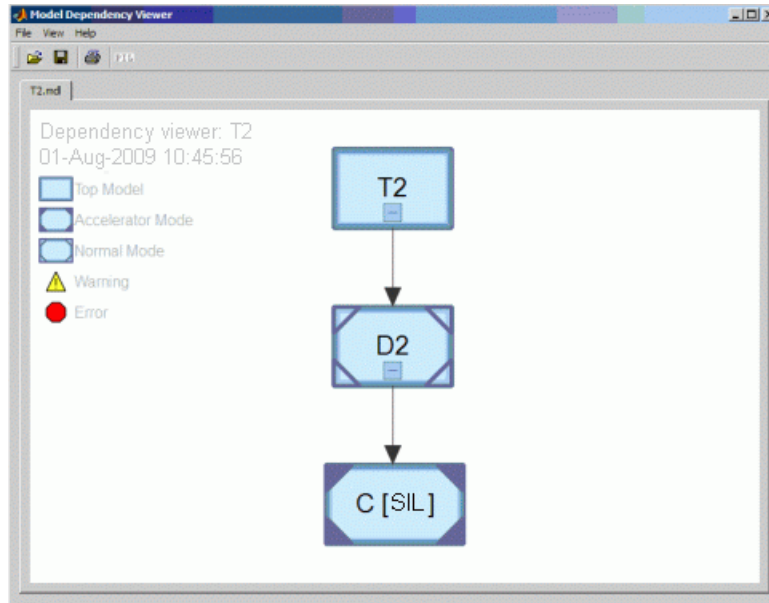
To pass test inputs to component C (running in SIL mode):

- 1 Create a test harness model that references model D2 in normal mode, as shown by model T2.



- 2 Simulate the T2 model to run component C in SIL mode and test its Real-Time Workshop model reference target.

The Model Dependency Viewer shows the model reference hierarchy of T2 and the simulation modes of each Model block component.



When to Use the SIL Block

Use the SIL block if you want to:

- Verify code generated for a top model (standalone code interface) or subsystem (right-click build standalone code interface). For more information about the standalone target code interface, see “SIL Code Interfaces” on page 36-16.
- Change the model and insert a SIL block to represent a component running in SIL mode, in a situation where a test harness model or a system model provides test vector or stimulus inputs.

Note If you compare a SIL block simulation with a top-model simulation, you see that you must perform two steps before you can run the simulation. First, do a right-click subsystem build to create the SIL block. Then, replace the subsystem in the original model with the newly created SIL block.

Running a Complete Model as a SIL Simulation

With a top-model SIL simulation:

- Simulink generates and executes code that uses the same code interface produced by the standalone build process. See “SIL Code Interfaces” on page 36-16.
- You can specify external stimulus signals and log output signals, which allows you to verify object code generated from a complete model without creating a separate test harness model. Running the SIL simulation is a simple single-step operation. See “Configuring a Top-Model PIL Simulation” on page 39-14.

Top-model SIL simulation is an alternative to the block-based approach where you provide a test harness model that wraps a Model block (in SIL mode). Two differences between the block-based approach and top-model SIL simulation are:

- With Model block SIL simulation, the model reference target that is generated does not have the same interface as standalone code (see “SIL Code Interfaces” on page 36-16).
- With Model block SIL simulations, you cannot directly specify external stimulus signals or enable signal logging. You must use input and output blocks to feed signals into and out of your model. See “Verifying Internal Signals of a Component” on page 36-14 and “Choosing a SIL Approach” on page 36-5.

To compare all SIL simulation options, see “Choosing a SIL Approach” on page 36-5.

For the top-model SIL approach, Simulink creates a hidden *wrapper* model. When you run a top-model SIL simulation, the software generates code for the model and creates a hidden wrapper model to call this code at each time step.

- If there are errors during a SIL simulation, you may see messages that refer to the wrapper model. For wrapper model error messages, the wrapper model is made visible to allow you to investigate the error.

- For *signal logging*, the software adds the suffix `_wrapper` to the block path for signals in `logouts`, as shown in the following example:

```
>> logouts.SignalLogging

      Name: 'SignalLogging'
      BlockPath: 'sillogging_wrapper/sillogging'
      PortIndex: 1
      SignalName: 'SignalLogging'
      ParentName: 'SignalLogging'
      TimeInfo: [1x1 Simulink.TimeInfo]
      Time: [11x1 double]
      Data: [11x1 double]
```

- For *output logging*, if the save format is `Structure` or `Structure with time`, the software adds the suffix `_wrapper` to the block name for signals in `yout`, as shown in the following example:

```
>> yout.signals

ans =
      values: [11x1 double]
      dimensions: 1
      label: 'SignalLogging'
      blockName: 'sillogging_wrapper/OutputLogging'
```

If the save format is `Array`, then the software does not add a wrapper suffix.

Running a Referenced Model as a SIL Simulation

In addition to the regular simulation modes, Model blocks have a Software-in-the-loop (SIL) and Processor-in-the-loop (PIL) mode.

You can switch the Model block between regular, SIL, and PIL simulation modes. This allows you to easily verify the generated code by executing the referenced model as compiled code on the host computer or target platform. You can model and test your embedded software component in Simulink and you can reuse your regression test suites across simulation and compiled object code. This capability avoids the time-consuming process of leaving the Simulink software environment to run tests again on object code compiled for your production hardware.

When a Model block is in SIL mode, you see the label (SIL) on the block.

To understand how SIL works in the Model block, see the following information:

- “Simulation Mode Override Behavior in Model Reference Hierarchy” on page 36-15
- “SIL Code Interfaces” on page 36-16
- “When to Use Model Block SIL” on page 36-6
- “Modeling Scenarios with the Model Block” on page 36-6

For an introduction to the Model block, see the Model block section in the Simulink reference documentation.

Verifying Internal Signals of a Component

Outputs of the SIL component are available for verification. If you want to examine an internal signal, you can manually route the signal to the top level.

Alternatively, you can use global data stores to access internal signals:

- 1 Inside the SIL component, connect a Data Store Write block to the required signal.

- 2 Outside the SIL component, use a Data Store Read block to access the signal value.

See “Working with Data Stores” and “Global Data Store Example”.

For more information on signal support, see “I/O Support” on page 39-33.

Simulation Mode Override Behavior in Model Reference Hierarchy

You can view your model hierarchy in the Model Dependency Viewer.

In the Referenced Model Instances view, Model blocks are displayed differently to indicate Normal, Accelerator, SIL, and PIL modes. A parent model can override the simulation mode of a Model block, as follows:

- A Model block in Accelerator mode overrides the behavior of any Model blocks beneath it in the model reference hierarchy. The Accelerator mode Model block links with the simulation targets (SIM targets) of the blocks beneath. See “Model Reference Simulation Targets” in the Simulink documentation.
- A Model block in SIL mode overrides the behavior of any Model blocks beneath it in the model reference hierarchy. The SIL mode Model block uses the model reference targets of the blocks beneath. See “Code Interface for Model Block SIL” on page 36-16.

If a block has a Normal mode path from the top model, the block behaves as specified by its simulation mode.

Note In a Model reference hierarchy, only one branch can simulate in SIL mode.

For an example model hierarchy, see “Modeling Scenarios with the Model Block” on page 36-6.

SIL Code Interfaces

This section describes the different code interfaces that the code generation products produce.

You generate “standalone code” when you perform a top-model or right-click subsystem build for a single deployable component. You can compile and link standalone code into a standalone executable or integrate it with other code. For more information on the standalone code interface, see “Model Entry Points” on page 31-14.

When you generate code for a referenced model hierarchy, the software generates standalone executable code for the top model, and a library module called a *model reference target* for each referenced model. When the code executes, the standalone executable invokes the applicable model reference targets to compute the referenced model outputs. For more information, see “Creating Model Components” in the Real-Time Workshop documentation.

Note The model reference target does not have the same code interface as standalone code.

If you intend to integrate automatically generated code with legacy code, use standalone code because the standalone code interface (for example, entry points) is fully documented.

Code Interface for Top-Model SIL

Top-model SIL generates the *standalone code interface* for the model.

When you run a top-model SIL simulation, the software calls the standalone code for the model if it already exists. The software generates the standalone code if it does not exist.

Code Interface for Model Block SIL

Model block SIL mode generates the *model reference* code interface.

When you run a simulation with a Model block in SIL mode, the software calls the model reference target for the Model block if it already exists, or generates the model reference target.

If the model reference target does not yet exist, you can generate it in one of three ways:

- Run the simulation.
- Press **Ctrl+B** to build the top model containing the Model block.
- Use the command `slbuild`, specifying the model reference option, for example:

```
slbuild('model', 'ModelReferenceRTWTargetOnly')
```

You cannot use standalone code with the Model block. You can generate standalone code for a model referenced by a Model block by opening the model and performing a top-level build. However, you cannot use this standalone code with Model block SIL simulation.

For more information, see the table in “Choosing a SIL Approach” on page 36-5.

Configuring a SIL Simulation

In this section...

- “Configuring a Top-Model SIL Simulation” on page 36-18
- “Configuring a Model Block SIL Simulation” on page 36-20
- “Using a Code Coverage Tool in a SIL Simulation” on page 36-22
- “Verifying a SIL or PIL Configuration” on page 36-26
- “Compatible Models” on page 36-28

Configuring a Top-Model SIL Simulation

To configure and run a top-model SIL simulation:

- 1 Open your model.
- 2 Select **Simulation > Software-in-the-Loop (SIL)**.

Note This option is available only if the model is configured for an ERT or AUTOSAR target. See “Real-Time Workshop Pane: General” and Chapter 22, “Generating Code for AUTOSAR Software Components” for configuration information.

- 3 If you have not already done so, in the Configuration Parameters dialog box, on the **Data Import/Export** pane:

- In the **Input** check box and field, specify stimulus signals (or test vectors) for your top model.
- Configure logging for model outputs, using either *output logging* or *signal logging*:
 - In the **Output** check box and field, specify *output logging*.
 - In the **Signal logging** check box and field, specify *signal logging*.

The software logs only signals that connect to root-level inports and outports. See “Verifying Internal Signals of a Component” on page 36-14. If the root outports of your model connect to bus signals, then

output logging is not available. Use signal logging for bus signals that connect to root outputs.

You can log signals that connect to inports or outputs of the top model if you name the signals. If you select **Signal logging** but do not name signals that connect to inports or outputs of the top model, then the signal logging object (for example, `logout`) does not hold signal data.

- Disable logging of Data Store Memory variables. The software does not support this option for this simulation mode. If you do not clear the **Data stores** check box, the software produces a warning when you run the simulation.

For information about the **Data Import/Export** pane, see “Importing and Exporting Data” and “Data Import/Export Pane”.

4 Specify one of the following:

- Hardware implementation settings that correspond to the host machine. See “Configuring the Hardware Implementation” in the Real-Time Workshop documentation.

For a SIL simulation, you do not have to specify a value for the **Byte ordering** field on the **Hardware Implementation** pane. The software uses the byte ordering that the host computer uses.

- Portable word sizes. See “Configuring Hardware Implementation Settings” on page 36-31.

5 If required, configure code coverage. See “Using a Code Coverage Tool in a SIL Simulation” on page 36-22.

6 Start the simulation.

Note You cannot:

- Close the model while the simulation is running. To interrupt the simulation, in the Command Window, press **Ctrl+C**.
 - Alter the model during the simulation. You can move blocks and lines as long as it does not alter the behavior of the model.
-

You can run a top-model SIL simulation using the command `sim(model)`.

Note The software supports the `sim` command options `SrcWorkspace` and `DstWorkspace` for only the following values:

- `SrcWorkspace` — 'base'
- `DstWorkspace` — 'base' or 'current'

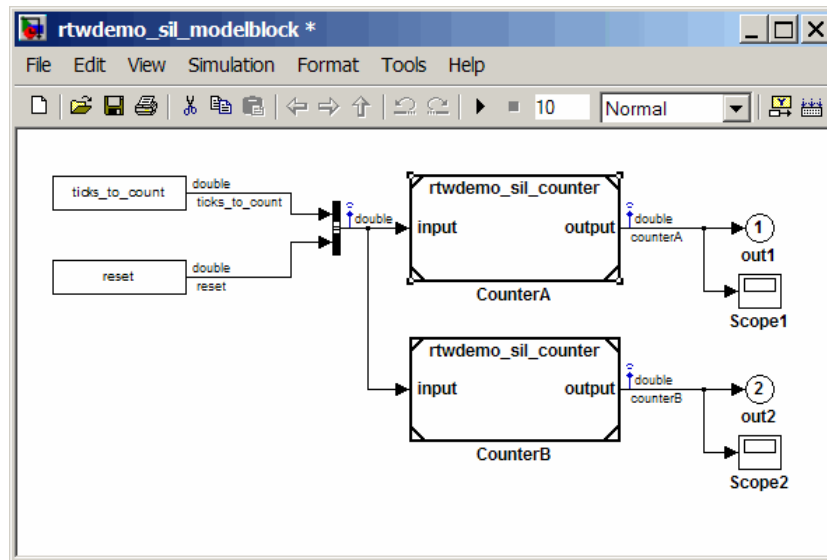
For more information on the `sim` command and its options, see “Simulation” in the Simulink documentation.

When running a top model in SIL simulation mode, the software ignores the Simulation mode setting for any Model block within the top model. For example, if the top model contains a Model block, the simulation mode of this block may be set to 'Normal', 'Accelerator', 'SIL' or 'PIL', with no effect on the top-model SIL simulation.

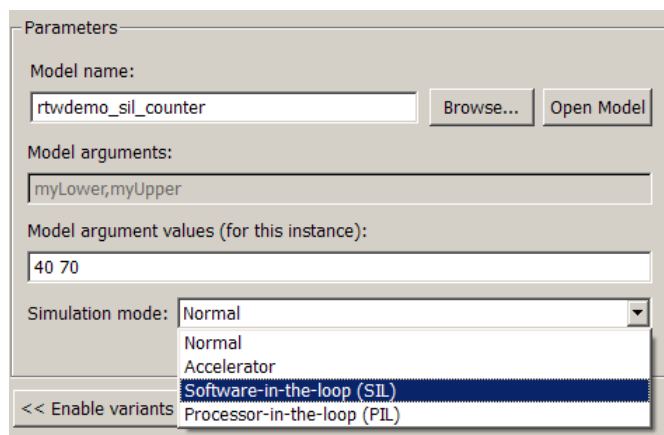
Configuring a Model Block SIL Simulation

To configure a Model block for a SIL simulation:

- 1 Open your model.



- 2 Right-click your Model block, for example, Counter A. In the context menu, select **ModelReference Parameters** to open the Model Reference Parameters dialog box.
- 3 From the **Simulation Mode** drop-down list, select Software-in-the-loop (SIL).



4 Select one of the following:

- Hardware implementation settings that correspond to the host machine. See “Configuring the Hardware Implementation” in the Real-Time Workshop documentation.

For a SIL simulation, you do not have to specify a value for the Byte ordering field on the Hardware Implementation pane. The software uses the byte ordering that the host computer uses.

- Portable word sizes. See Setting Up a Model to Generate Code for Host Simulations and Target Deployment

5 Start the simulation.

Using a Code Coverage Tool in a SIL Simulation

During a top-model or Model block SIL simulation, you can collect code coverage metrics for generated code using a third-party tool.

Real-Time Workshop Embedded Coder supports the BullseyeCoverage™ tool from Bullseye Testing Technology™. The MathWorks does not supply this tool. For information about installing and using this tool, go to <http://www.bullseye.com/cgi-bin/mwEval>.

To configure a code coverage tool for a top-model or Model block SIL simulation:

1 Select **Simulation > Configuration Parameters > Real-Time Workshop > SIL and PIL Verification**

2 From the **Code coverage tool** drop-down list, select a tool, for example, BullseyeCoverage.

3 Click **Configure Coverage** to open the Code Coverage Settings dialog box.

4 In the **Installation folder** field, specify the location where your coverage tool is installed.

5 Specify the models for which you want code coverage data:

- To generate coverage data for just the current (top) model, select the **Code coverage for this model** check box.

- To generate data for models referenced by the current (top) model, select the **Code coverage for referenced models** check box.

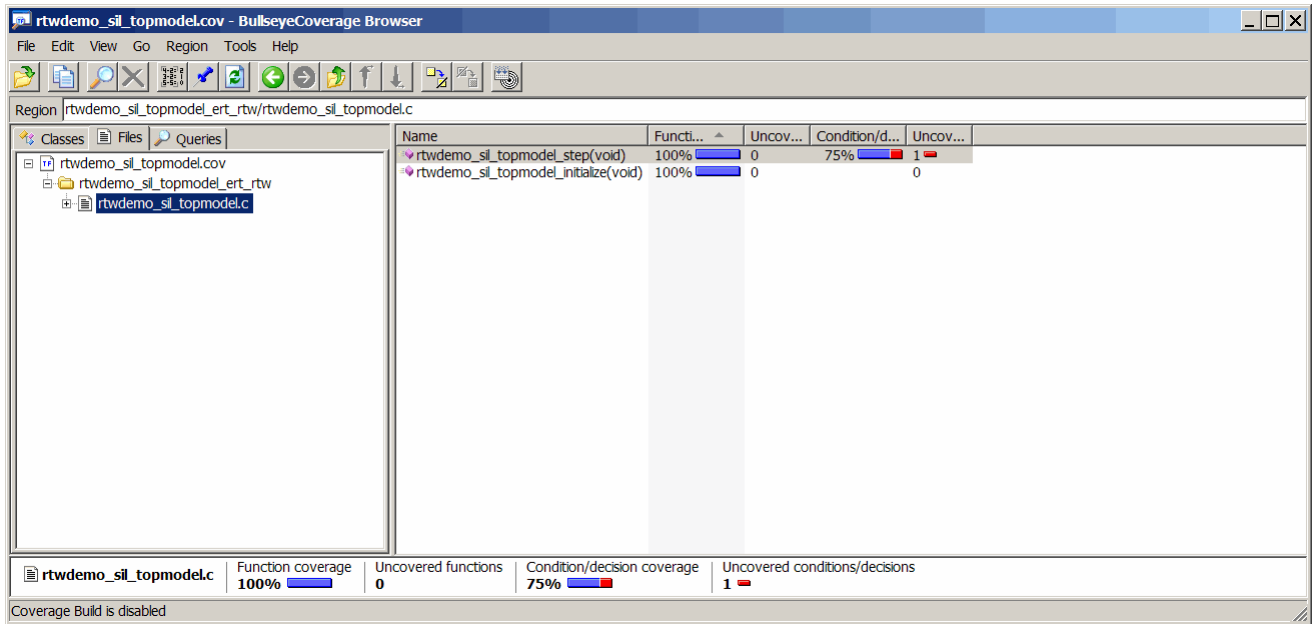
Note If you do not select a check box, the software does *not* generate code coverage data.

- 6** Click **OK**. You return to the **SIL and PIL Verification** pane.
- 7** Click **OK**. You return to the model window.
- 8** Start the SIL simulation. See “Configuring a Top-Model SIL Simulation” on page 36-18 and “Configuring a Model Block SIL Simulation” on page 36-20.

When the SIL simulation is complete, you see a hyperlink in the Command Window.

```
### Starting Real-Time Workshop build procedure for model: rtwdemo_sil_topmodel
### Successful completion of Real-Time Workshop build procedure for model: rtwdemo_sil_topmodel
### Preparing to start SIL simulation ...
### Starting SIL simulation for model: rtwdemo_sil_topmodel
### Stopping SIL simulation for model: rtwdemo_sil_topmodel
Code coverage data was collected; you can open the code coverage report in the BullseyeCoverage browser.
>>
```

Click this link to open the coverage report. For example, with the BullseyeCoverage Browser, you see the following window.



For information about the BullseyeCoverage Browser, go to www.bullseye.com

Tips and Limitations

Compilers and Platforms Supported for Code Coverage. For code coverage, the software supports the following compilers and platforms:

- On a Linux® platform, gcc
- On a Windows platform, vc

The software does *not* support the Lcc or Watcom compiler. If either compiler is specified with code coverage, the software produces an error when you build your model. For information on how to specify a compiler, see “Choosing and Configuring a Compiler” in the Real-Time Workshop documentation.

Characters in matlabroot and File Path. If matlabroot or the path to your generated files contains a space or the . (period) character, code coverage fails.

Code Coverage Instrumentation in a Standalone Executable. If you build your model and the model is configured for code coverage, then the generated standalone executable contains instrumentation for collecting code coverage metrics. However, MathWorks recommends that you use only the SIL simulation mode to generate code coverage metrics.

.cov Files in Build Folders. The software uses only the .cov file in the code generation working folder to collect code coverage metrics for the entire model hierarchy. You can see other .cov files in the build folders for individual model components, for example, the top model and referenced models. However, these files are intermediate files, which the software does not use during a SIL simulation.

Code Coverage for Source Files in Shared Utility Folders. The software supports code coverage for source files generated in shared utility folders. If you configure code coverage for a model that uses shared utility function generation, when you build the model, you also build all source files in the shared utilities folder with code coverage enabled.

Whenever you build a model, the code coverage settings of the model must be consistent with source files that you previously built in the shared utilities folder. Otherwise, the software reports that code in the shared utilities folder is inconsistent with the current model configuration and must be rebuilt. For example, if you run a SIL simulation for a model with code coverage enabled and then a SIL simulation for another model with code coverage disabled, the software must rebuild all source files in the shared utilities folder.

The BullseyeCoverage tool, by default, does not provide code coverage data for inline macros.

For example, if a model generates a file `slprj/ert/_sharedutils/rt_SATURATE.h` that contains the macro

```
#define rt_SATURATE(sig,ll,u1) (((sig) >= (u1)) ? (u1) :  
(((sig) <= (ll)) ? (ll) : (sig)) )
```

and the macro is in `sat_ert_rtw/sat.c`, then the coverage report provides a measurement for `sat.c`, but no coverage data for the conditions within the macro `rt_SATURATE`.

To configure the `BullseyeCoverage` tool to provide code coverage data for inline macros:

- 1 Open the `BullseyeCoverage` Browser.
- 2 Select **Tools > Options** to open the Options dialog box.
- 3 On the **Build** tab, select the **Instrument macro expansions** check box.
- 4 Click **OK**.
- 5 Rerun your simulation.

Alternatively, you can add the text `-macro` to the `BullseyeCoverage` configuration file. For more information, go to http://www.bullseye.com/help/ref_covc.html.

Verifying a SIL or PIL Configuration

You might need to change model settings to configure the model correctly for SIL or PIL. To find out what settings you must change, use the `cgv.Config` class. Using the `cgv.Config` class, you can review your model configuration and determine which settings you must change to configure the model correctly for SIL or PIL. By default, `cgv.Config` changes configuration parameter values to the value that it recommends, but does not save the model. Alternatively, you can specify that `cgv.Config` use one of the following approaches:

- Change configuration parameter values to the values that `cgv.Config` recommends, and save the model. Specify this approach using the `SaveModel` property.
- List the values that `cgv.Config` recommends for the configuration parameters, but do not change the configuration parameters or the model. Specify this approach using the `ReportOnly` property.

Note

- To execute the model in the target environment successfully, you might need to make additional modifications to the configuration parameter values or the model.
 - Do not use referenced configuration sets in models that you are changing using `cgv.Config`. If the model uses a referenced configuration set, update the model with a copy of the configuration set. Use the `Simulink.ConfigSetRef.getRefConfigSet` method. For more information, see `Simulink.ConfigSetRef` in the Simulink documentation.
 - If you use `cgv.Config` on a model that executes a callback function, the callback function might change configuration parameter values each time the model loads. The callback function might revert changes that `cgv.Config` made. When this change occurs, the model might no longer be set up correctly for SIL or PIL. For more information, see “Using Callback Functions”.
-

For more information about the `cgv.Config` class, see `cgv.Config`.

How To Verify a SIL or PIL Configuration

To verify that your model is configured correctly:

- 1 Construct a `cgv.Config` object that changes the configuration parameter values without saving the model. For example, to configure your model for SIL:

```
c = cgv.Config('vdp', 'connectivity', 'sil');
```

Tip

- You can obtain a list of changes without changing the configuration parameter values. When you construct the object, include the 'ReportOnly', 'on' property name and value pair.
 - You can change the configuration parameter values and save the model. When you construct the object, include the 'SaveModel', 'on' property name and value pair.
-

- 2 Determine and change the configuration parameter values that the object recommends using the `configModel` method. For example:

```
c.configModel();
```

- 3 Display a report of the changes that `configModel` makes. For example:

```
c.displayReport();
```

- 4 Review the changes.

- 5 To apply the changes to your model, save the model.

Compatible Models

You can use the SIL simulation mode with any Simulink model, if:

- 1 The model specifies an ERT-based target.
- 2 The model specifies one of the following template makefiles:

- a `ert_default_tmf`
- b `ert_unix.tmf`
- c `ert_vc.tmf`
- d `ert_vcx64.tmf`
- e `ert_lcc.tmf`

The software does not support the Watcom compiler template makefile (`ert_watc.tmf`).

3 The host and target word sizes match. For example, if your host is a Windows XP computer, then you must specify, through the **Hardware Implementation** pane, a hardware device with the following word sizes (in bits):

- **char** — 8
- **short** — 16
- **int** — 32
- **long** — 32
- **native word size** — 32

See also “Configuring Hardware Implementation Settings” on page 36-31.

Verifying Code Using a SIL Block

When using the Real-Time Workshop product to set up SIL tests, you write an S-function wrapper and use the `mex` command to create a block to represent the S-function. If you have a Real-Time Workshop Embedded Coder license, the setup procedure for SIL tests is simpler.

- 1** In the Configuration Parameters dialog box, select **Real-Time Workshop > SIL and PIL Verification > Create SIL block**.
- 2** Right-click the subsystem that you want to simulate on the MATLAB host system.
- 3** Select **Real-Time Workshop > Build Subsystem**. This initiates a subsystem build that generates an S-function wrapper and block for the generated subsystem code.
- 4** Add the generated S-function block to an environment or test harness model that supplies test vectors or stimulus input.
- 5** Run SIL tests on the host system by simulating the environment model.
- 6** Verify that the generated code captured in the SIL block provides the same result as the original subsystem.

For an example of SIL testing, see `rtwdemo_sil_pil_script`. For more information about using the **Create SIL block** parameter to generate an S-function wrapper, including limitations, see Chapter 24, “Generating S-Function Wrappers”.

Configuring Hardware Implementation Settings

In this section...
“Compiling Generated Code That Supports Portable Word Sizes” on page 36-33
“Portable Word Sizes Limitation” on page 36-33

Real-Time Workshop Embedded Coder provides an option to specify portable word sizes. If you select this option for a model, you can use the same generated source code files for:

- Software-in-the-loop (SIL) simulation on the host computer
- Production deployment on the target platform

If you do not specify portable word sizes for a SIL simulation, you can configure the model to use an emulation hardware option. For integer and fixed-point operations, this option guarantees bit-true agreement between host computer simulation and target deployment. See “Configuring the Hardware Implementation” in the Real-Time Workshop documentation.

Note If processor word sizes differ between host and target platforms, and you do not use any of the preceding options for SIL simulation, there are likely to be differences between host computer results and target execution results. When you use portable word sizes for SIL simulation, subtle differences in host and target processor behavior can still occasionally cause host simulation results to differ from target execution results. For more information, see “Portable Word Sizes Limitation” on page 36-33.

To configure a model to use portable word sizes, set the following model configuration parameters.

Set...	To...
Hardware Implementation > Emulation hardware > None	Selected
Real-Time Workshop > SIL and PIL Verification > Create SIL block	Selected
Real-Time Workshop > SIL and PIL Verification > Enable portable word sizes	Selected

When you generate code for a model with the preceding parameter settings, the code generator conditionalizes data type definitions:

- `tmwtypes.h` supports SIL simulation on the host system
- Real-Time Workshop types support deployment on the target system

For example, in the following generated code, the first two lines define types for SIL simulation on a host system. The **bold** lines define types for target deployment.

```

#ifdef PORTABLE_WORDSIZES      /* PORTABLE_WORDSIZES defined */
# include "tmwtypes.h"
#else                          /* PORTABLE_WORDSIZES not defined */
#define __TMWTYPES__
#include <limits.h>
...
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef int int16_T;
typedef unsigned int uint16_T;
typedef long int32_T;
typedef unsigned long uint32_T;
typedef float real32_T;
typedef double real64_T;
...
#endif                          /* PORTABLE_WORDSIZES */

```

For an example of how to configure a model to maintain bit-true agreement between host simulation and target deployment, and generate code that is portable between the host and target systems, see `rtwdemo_sil_hardware_config`.

Compiling Generated Code That Supports Portable Word Sizes

When you compile generated code that supports portable word sizes for SIL testing, pass the definition `PORTABLE_WORDSIZES` to the compiler.

For example:

```
-DPORTABLE_WORDSIZES
```

To build the same code for target deployment, compile the code without the `PORTABLE_WORDSIZES` definition.

Portable Word Sizes Limitation

When you perform SIL testing, using the **Enable portable word sizes** model configuration parameter, numerical results of the S-function simulation on the MATLAB host might differ from results on the actual target. This difference is due to differences in target characteristics, such as:

- C integral promotion in expressions may be different on the target processor
- Signed integer division rounding behavior may be different on the target processor
- Signed integer arithmetic shift right may behave differently on the target processor
- Floating-point precision may be different on the target processor

SIL Support and Limitations

For information about SIL support and limitations, see “SIL and PIL Simulation Support and Limitations” on page 39-24.

Verifying a Component in the Target Environment

- “About Component Verification in the Target Environment” on page 37-2
- “Goals of Component Verification in the Target Environment” on page 37-3
- “Maximizing Code Portability and Configurability” on page 37-4
- “Simplifying Code Integration and Maximizing Code Efficiency” on page 37-5
- “Running Component Tests in the Target Environment” on page 37-7

About Component Verification in the Target Environment

After you generate production code for a component design, you need to integrate, compile, link, and deploy the code as a complete application on the embedded system. One approach is to manually integrate the code into an existing software framework that consists of an operating system, device drivers, and support utilities. The algorithm can include externally written legacy or custom code.

An easier and more recommended approach to verifying a component in a target environment, is to use processor-in-the-loop (PIL) simulation. For details on applying PIL simulations, see Chapter 39, “Verifying Compiled Object Code with Processor-in-the-Loop Simulation”.

Goals of Component Verification in the Target Environment

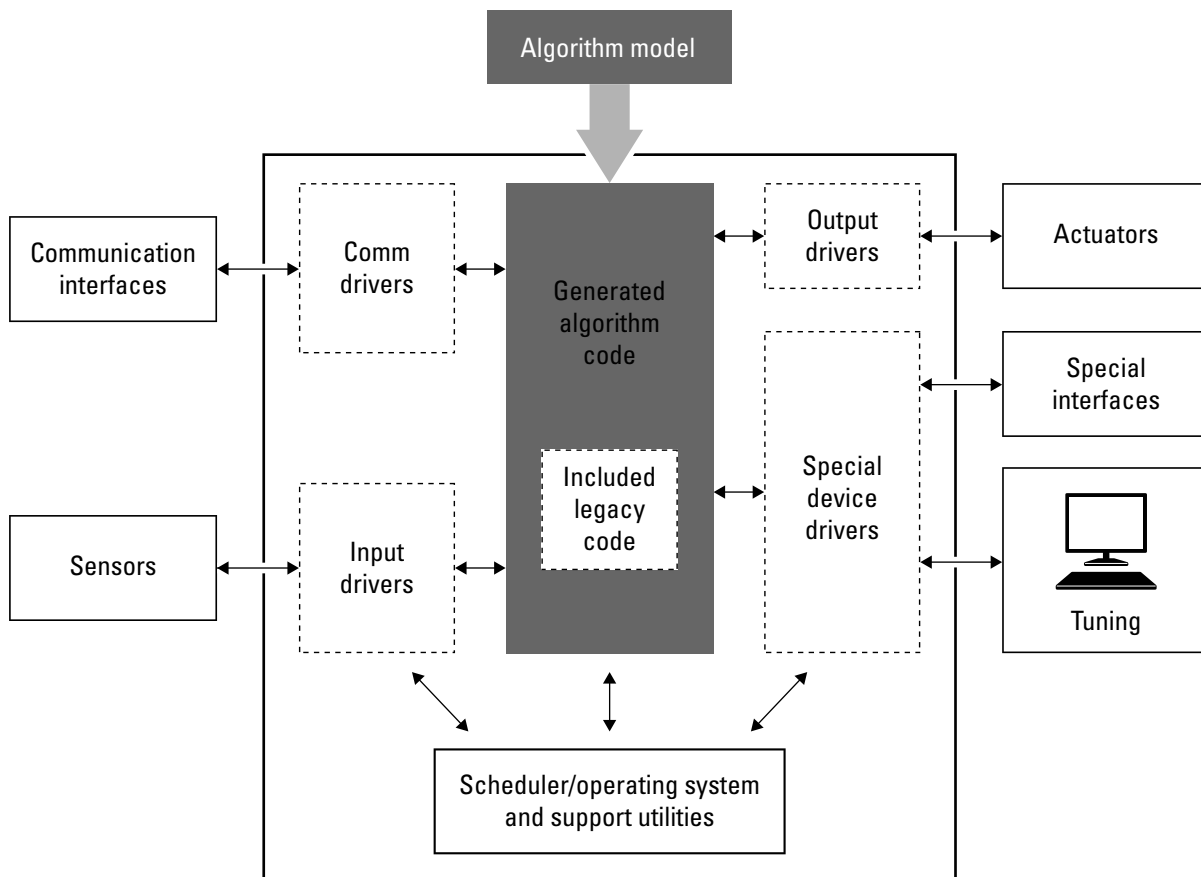
Assuming that you have generated production quality source code and integrated necessary externally written code, such as drivers and a scheduler, you can verify that the integrated software operates correctly by testing it in the target environment. During testing, you can achieve either of the following goals, depending on whether you export code that is strictly ANSI C/C++ or mixes ANSI C/C++ with code optimized for a target environment.

Goal	Type of Code Export
Maximize code portability and configurability	ANSI C/C++
Simplify integration and maximize use of processor resources and code efficiency	Mixed code

Regardless of your goal, you must integrate any required external driver and scheduling software. To achieve real-time execution, you must integrate the necessary real-time scheduling software.

Maximizing Code Portability and Configurability

To maximize code portability and configurability, limit the application code to ANSI/ISO C or C++ code only, as the following figure shows.

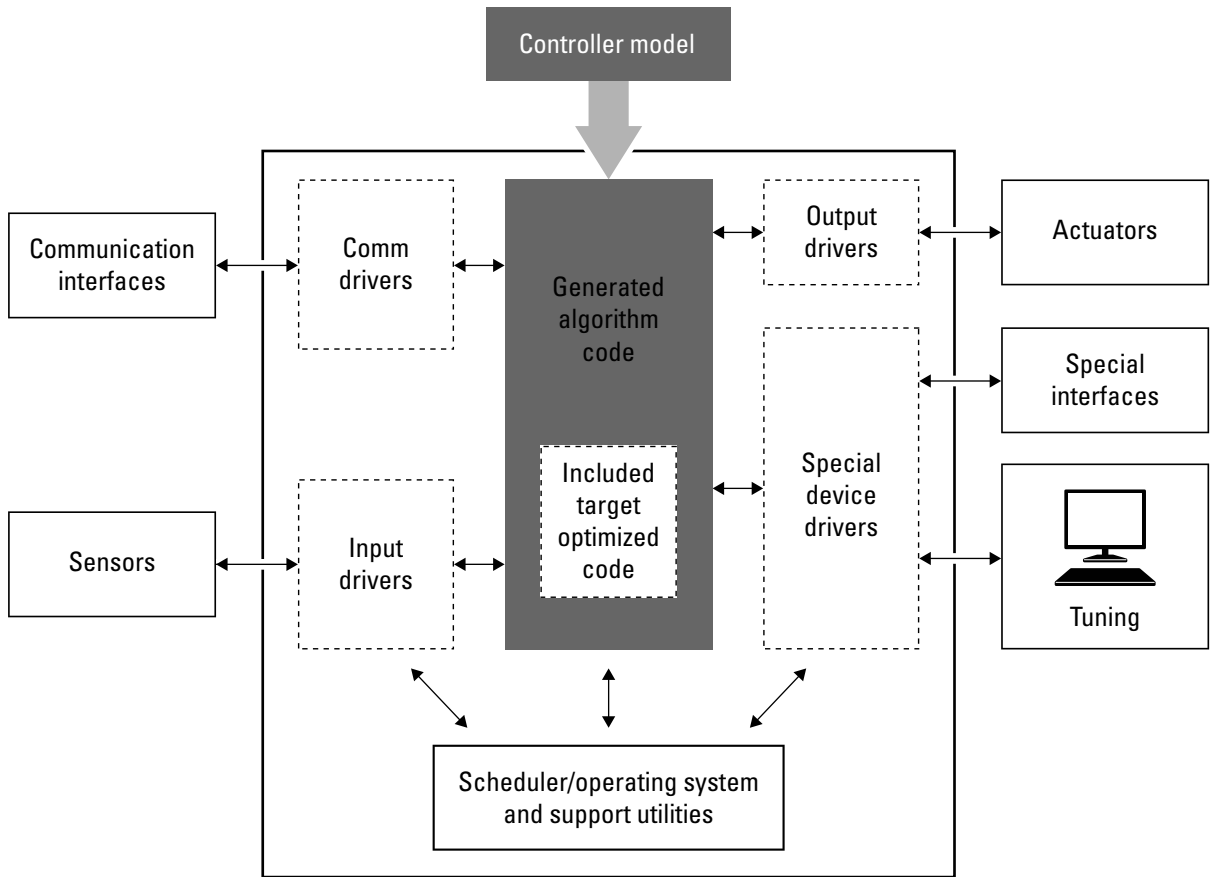


Simplifying Code Integration and Maximizing Code Efficiency

To simplify code integration and maximize code efficiency for a target environment, use Real-Time Workshop Embedded Coder features for:

- Controlling code interfaces
- Exporting subsystems
- Including target-specific code, including compiler optimizations

The following figure shows a mix of ANSI C/C++ code with code that is optimized for a target environment.



Running Component Tests in the Target Environment

The workflow for running software component tests in the target environment is:

- 1 Integrate external code, for example, for device drivers and a scheduler, with the generated C or C++ code for your component model. For more information, see Integrating External Code and Generated C and C++ Code on page 1 and “Integrating External Code With Generated C and C++ Code” in the Real-Time Workshop documentation. For more specific references depending on your verification goals, see the following table.

For...	See...
ANSI C/C++ code integration	“Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation. Also, open <code>rtwdemos</code> and navigate to the Custom Code folder.
Mixed code integration	<ul style="list-style-type: none"> • Chapter 25, “Exporting Function-Call Subsystems” and <code>examplertwdemo_export_functions</code> • Chapter 27, “Controlling Generation of Function Prototypes”, Chapter 28, “Controlling Generation of Encapsulated C++ Model Interfaces”, and example <code>rtwdemo_fcnprotoctrl</code> • Chapter 29, “Replacing Math Functions and Operators Using Target Function Libraries” and example <code>rtwdemo_tf1_script</code>

- 2 Simulate the integrated component model.
- 3 Generate code for the integrated component model.
- 4 Connect to data interfaces for the generated C code data structures. See “Interacting with Target Application Data Using the C API” and “Generating Model Information for Host-Based ASAP2 Data Measurement

and Calibration” in the Real-Time Workshop documentation. Also see examples `rtwdemo_capi` and `rtwdemo_asap2`.

- 5** Customize and control the build process, as necessary. See “Customizing Code Generation and the Build Process”, in the Real-Time Workshop documentation, and example `rtwdemo_buildinfo`.
- 6** Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer. See “Relocating Code to Another Development Environment”, in the Real-Time Workshop documentation, and example `rtwdemo_buildinfo`.

Verifying a Component by Building a Complete Real-Time Target Environment

- “About Component Verification With a Complete Real-Time Target Environment” on page 38-2
- “Goals of Component Verification With a Complete Real-Time Target Environment” on page 38-4
- “Testing a Component as Part of a Complete Real-Time Target Environment” on page 38-5

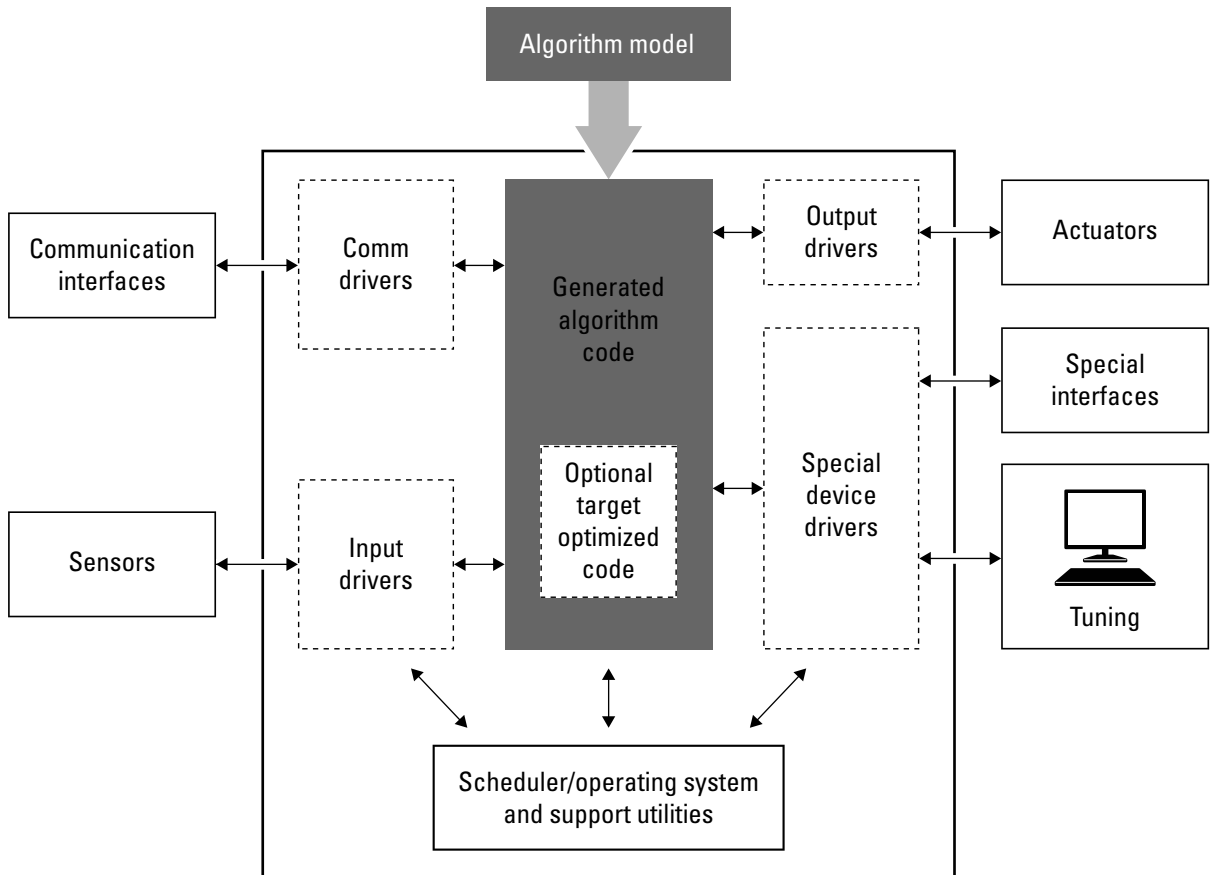
About Component Verification With a Complete Real-Time Target Environment

One approach to verifying a software component is to build the component into a complete software system that can execute in real time in the target environment. A complete software system includes:

- Algorithm for the software component
- Scheduling algorithms
- Calls to drivers for board-specific devices

This single build approach is more time consuming to set up, but makes it easier to get the complete application running in the target environment.

The following figure shows code generated for an algorithm being built into a complete system executable for the target environment.



Goals of Component Verification With a Complete Real-Time Target Environment

Assuming that you have generated production quality source code and integrated necessary externally written code, such as drivers and a scheduler, you can verify that component software operates correctly in the context of a complete system for testing in the target environment.

Testing a Component as Part of a Complete Real-Time Target Environment

The workflow for testing component software as part of a complete real-time target environment is:

1 Develop a component model and generate source code for production.

For information on building in scheduling and real-time system support, see:

- “Scheduling Considerations” in the Real-Time Workshop documentation. For an example, open `rtwdemos` and navigate to the **Multirate Support** folder.
- “Handling Asynchronous Events” in the Real-Time Workshop documentation and example `rtwdemo_async`
- “Interfacing With a Real-Time Operating System ” in the Real-Time Workshop documentation and Chapter 32, “Wind River Systems VxWorks Example Main Program”
- Chapter 31, “Interfacing With Hardware That is Not Running an Operating System (Bare Board)”
- Chapter 22, “Generating Code for AUTOSAR Software Components” and examples `rtwdemo_autosar_legacy_script`, `rtwdemo_autosar_multirunnables_script`, and `rtwdemo_autosar_clientserver_script`
- Example `rtwdemo_osek`

2 Optimize generated code for a specific run-time environment, using specialized function libraries. For more information, see Chapter 29, “Replacing Math Functions and Operators Using Target Function Libraries” and example `rtwdemo_tfl_script`.

3 Customize post code generation build processing to accommodate third-party tools and processes, as necessary. See “Customizing Code Generation and the Build Process” in the Real-Time Workshop documentation and example `rtwdemo_buildinfo`.

- 4** Integrate external code, for example, for device drivers and a scheduler, with the generated C or C++ code for your component model. For more information, see Integrating External Code and Generated C and C++ Code on page 1 and “Integrating External Code With Generated C and C++ Code” in the Real-Time Workshop documentation. For more specific references depending on your verification goals, see the following table.

For...	See...
ANSI C/C++ code integration	“Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation. Also, open <code>rtwdemos</code> and navigate to the Custom Code folder.
Mixed code integration	<ul style="list-style-type: none"> • Chapter 25, “Exporting Function-Call Subsystems” and <code>examplertwdemo_export_functions</code> • Chapter 27, “Controlling Generation of Function Prototypes”, Chapter 28, “Controlling Generation of Encapsulated C++ Model Interfaces”, and example <code>rtwdemo_fcnprotoctrl</code> • Chapter 29, “Replacing Math Functions and Operators Using Target Function Libraries” and example <code>rtwdemo_tfl_script</code>

- 5** Simulate the integrated model.
- 6** Generate code for the integrated model.
- 7** Connect to data interfaces for the generated C code data structures. See “Interacting with Target Application Data Using the C API” and “Generating Model Information for Host-Based ASAP2 Data Measurement and Calibration” in the Real-Time Workshop documentation. Also see examples `rtwdemo_capi` and `rtwdemo_asap2`.

- 8** Customize and control the build process, as necessary. See “Customizing Code Generation and the Build Process”, in the Real-Time Workshop documentation, and example `rtwdemo_buildinfo`.
- 9** Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer. See “Relocating Code to Another Development Environment”, in the Real-Time Workshop documentation, and example `rtwdemo_buildinfo`.

Verifying Compiled Object Code with Processor-in-the-Loop Simulation

- “About Processor-In-the-Loop Simulation” on page 39-2
- “Choosing a PIL Approach” on page 39-5
- “Programming PIL Support for Third-Party Tools and Target Hardware” on page 39-8
- “PIL Code Interfaces” on page 39-9
- “Setting Up PIL Simulations With the Embedded IDE Link Product PIL Block” on page 39-10
- “Configuring a PIL Simulation” on page 39-14
- “Creating a Connectivity Configuration for a Target” on page 39-18
- “SIL and PIL Simulation Support and Limitations” on page 39-24

About Processor-In-the-Loop Simulation

In this section...

“What Is Processor-in-the-Loop Simulation?” on page 39-2

“Why Use PIL Simulation?” on page 39-2

“Problems That PIL Simulation Can Detect” on page 39-3

“How PIL Simulation Works” on page 39-3

What Is Processor-in-the-Loop Simulation?

You can use processor-in-the-loop (PIL) simulation to verify your generated code. PIL is a technique that helps you evaluate the behavior of a candidate algorithm (for instance, a control or signal processing algorithm) on the target processor (or instruction set simulator) selected for the application. In PIL simulation, the target processor participates fully in the simulation loop — hence the term *processor-in-the-loop simulation*. You can compare the output of regular simulation modes, such as Normal or Accelerator, and PIL simulation mode to verify your generated code.

You can easily switch between regular simulation, SIL, and PIL modes. This flexibility allows you to verify the generated code by executing the model as compiled code in the target environment. You can model and test your embedded software component in Simulink and then reuse your regression test suites across simulation and compiled object code. This avoids the time-consuming process of leaving the Simulink software environment to run tests again on object code compiled for the production hardware.

Why Use PIL Simulation?

Normal simulation techniques do not account for restrictions and requirements that the hardware imposes, such as limited memory resources or behavior of target-specific optimized code. PIL simulation provides an intermediate stage between simulation and deployment. You can perform code verification and evaluate code performance earlier in the development cycle. When you reach the stage of deploying code on the target hardware, PIL allows you to test the object code using test vectors that you have developed and applied to your Simulink model. You can verify correct code behavior on

your target hardware and collect metrics, for example, code coverage and execution time. If you need to return to the original model and change it, you achieve faster iteration between model development and generated code verification.

Problems That PIL Simulation Can Detect

PIL is useful for detecting problems such as:

- Incorrect target-specific code
 - Undetectable with software-in-the-loop (SIL) testing because of the target-specific nature of the optimized code
- Unwanted side effects of compiler settings and optimizations
- Floating-point implementation issues
 - Floating-point applications may produce slightly different results in simulation and on hardware due to different floating-point implementations (unlike standard fixed-point applications, which give identical results in simulation and on hardware).
 - For example, the target may not implement strict IEEE® floating point. PIL detects these differences and allows you to analyze the differences.
 - In a closed-loop model, you can analyze build up of floating-point errors in the whole system.
- Code generator bugs
- Compiler bugs

You can also use PIL simulation with third-party tools, like a debugger with features such as code coverage, report generation, and execution profiling.

How PIL Simulation Works

In a PIL simulation, Real-Time Workshop Embedded Coder software generates code for either the top model or part of the model. This code is cross-compiled for the target hardware and runs on the target platform.

Simulink sends, via a communication channel, stimulus signals to the code on the target platform for each sample interval of the simulation:

- For a top model, Simulink uses stimulus signals from the base or model workspace.
- If you have designated only part of the model to simulate in PIL mode, then a part of the model remains in Simulink without the use of code generation. Typically, you configure this part of the model to provide test vectors for the software executing on the hardware. This part of the model can represent other parts of the algorithm or the environment model in which the algorithm operates.

When the target platform receives signals from Simulink, it executes the PIL algorithm for one sample step. The PIL algorithm returns output signals computed during this step to Simulink through a communication channel. At this point, one sample cycle of the simulation is complete and Simulink proceeds to the next sample interval. The process repeats and the simulation progresses. PIL simulations do not run in real time. At each sample period, Simulink and the object code exchange *all* I/O data. See also “Verifying Internal Signals of a Component” on page 36-14.

Choosing a PIL Approach

In this section...
“Comparing Methods for Applying PIL Simulations” on page 39-5
“Modeling Scenarios with the PIL Block” on page 39-5

Comparing Methods for Applying PIL Simulations

You choose a top-model PIL, Model block PIL, or PIL block verification approach in the same way that you choose the corresponding SIL approach. See “Choosing a SIL Approach” on page 36-5.

Note also that,

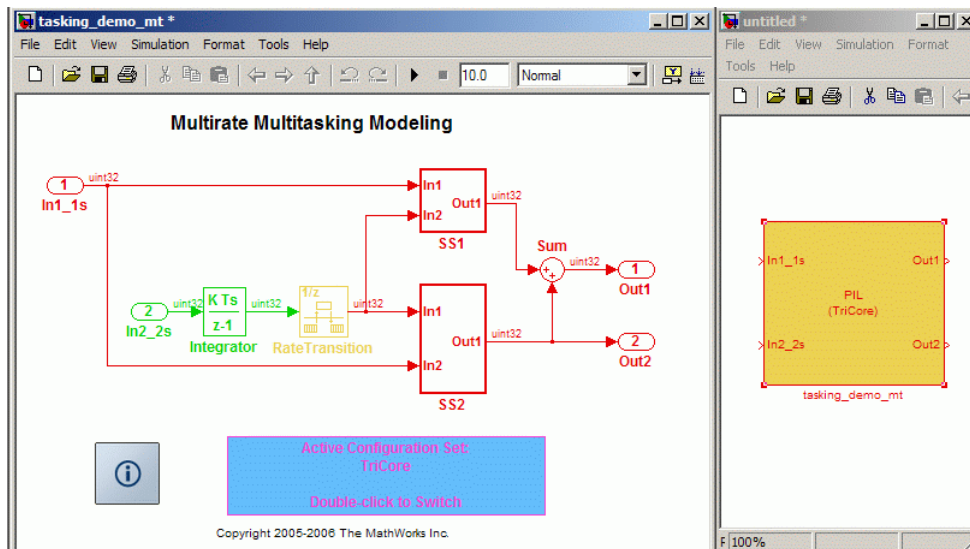
- The software provides Target Connectivity API support for only top-model and Model block PIL.
- The PIL block is only available with the Embedded IDE Link product.

Modeling Scenarios with the PIL Block

You can create a PIL block for a top-model build or a right-click build for a subsystem.

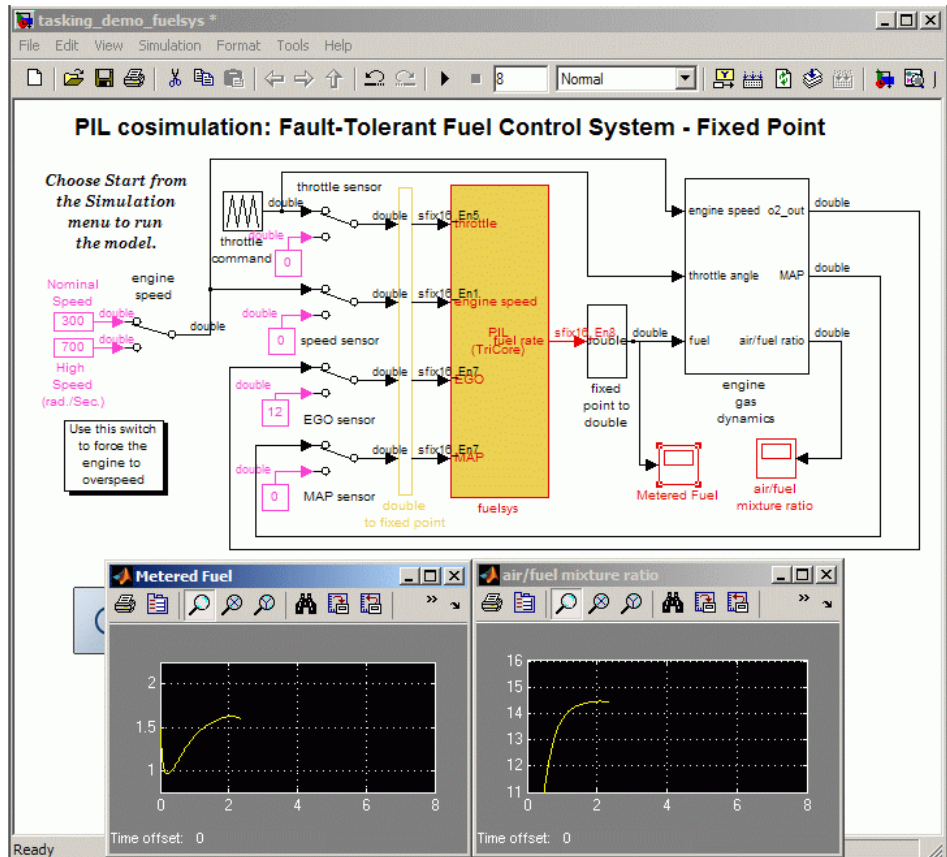
For a top-model build, you get a PIL block that represents the whole model. You can test the deployable standalone object code by calling the PIL block from a test harness model.

The following example shows the PIL block generated for a top-model build of the `tasking_demo_mt` model.



For a right-click subsystem build, you get a PIL block that represents the subsystem. You can test the deployable standalone object code, for example, by copying the PIL block into the original model to replace the original subsystem.

The following example shows the PIL block generated for the fuelsys subsystem, and copied into the original model.



Programming PIL Support for Third-Party Tools and Target Hardware

With the top-model and Model block PIL modes, you can use the Processor-in-the-loop (PIL) Connectivity API to apply the power of PIL verification to object code compiled for your target processor. There are many custom or third-party tools for building, downloading, and communicating with an executable on a target environment. Use the API to integrate your tools for:

- Building the PIL application (an executable for the target hardware)
- Downloading and running the executable
- Communicating with the executable

You can use PIL with any target hardware or instruction set simulator, and any combination of tools that provide the required level of automation. For hardware cases that The MathWorks does not support, see “SIL and PIL Simulation Support and Limitations” on page 39-24.

For instructions and demos on top-model and Model block PIL modes and the Target Connectivity API, see:

- “Configuring a PIL Simulation” on page 39-14
- “Creating a Connectivity Configuration for a Target” on page 39-18
- “Demos of the Target Connectivity API” on page 39-22

PIL Code Interfaces

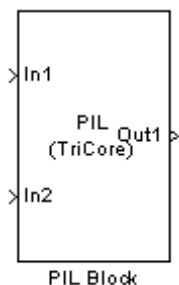
The code interfaces for top-model PIL, Model block PIL and PIL block are similar to those for the corresponding SIL modes. See “SIL Code Interfaces” on page 36-16.

With the Embedded IDE Link product, you can generate standalone code through a top-model or a right-click subsystem build, and automatically create a PIL block to test the code. Code is rebuilt only if necessary.

Setting Up PIL Simulations With the Embedded IDE Link Product PIL Block

The *PIL block* is available only with the Embedded IDE Link product. You cannot use the Target Connectivity API with the PIL block.

With the Embedded IDE Link product, you can automatically create a PIL block to test the code generated from your model. During the Real-Time Workshop Embedded Coder code generation process, you can create a PIL block from a complete model or from a subsystem.



After you create and build a PIL block, you can either:

- Copy the PIL block into your model to replace the original subsystem. Save the original subsystem in a different model so that you can restore it.
- Add the PIL block to your model to compare it to the original subsystem during simulation.

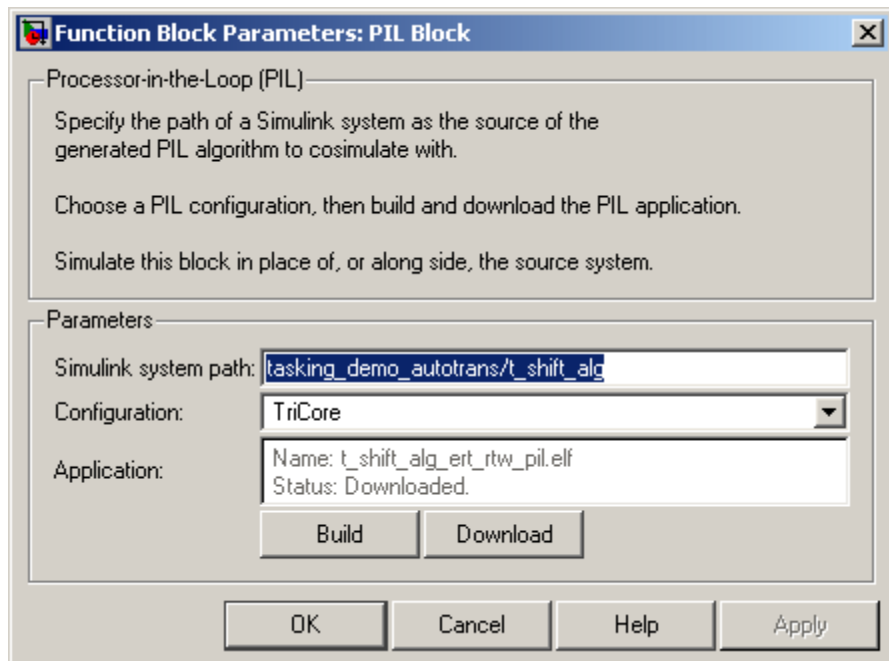
To understand the differences between Model block PIL and the PIL block, see:

- “SIL Code Interfaces” on page 36-16
- “Choosing a PIL Approach” on page 39-5
- “Modeling Scenarios with the PIL Block” on page 39-5
- “SIL and PIL Simulation Support and Limitations” on page 39-24

Note You can use the PIL block with Embedded IDE Link software. However, you can only use top-model PIL and Model block PIL with Embedded IDE Link software that is specific to Altium®TASKING®.

The PIL block is a basic building block that you can use to:

- Select a PIL algorithm
- Choose a PIL configuration
- Build and download a PIL application
- Run a PIL simulation



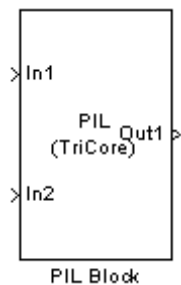
To build and download the PIL application manually:

- 1 Double-click the PIL block to open the mask.

- 2 Click **Build**. Wait until you see the updated **Application** name in the mask and the “build complete” message.
- 3 Click **Download**.
- 4 Wait until you see the “download complete” message in the PIL block. Click **OK** to close the block mask.

The PIL application is now ready. To run a PIL simulation with it, copy the PIL block into your model, either to replace the original subsystem or as an additional block for comparison. Click **Start Simulation**.

The PIL block takes the same shape and signal names as the parent subsystem. This inheritance is convenient for copying the PIL block into the model to replace the original subsystem for the PIL simulation.



The PIL block has the following parameters:

- **Simulink system path** — Allows you to select a PIL algorithm. You specify the path of a Simulink system (model or subsystem) as the source of the generated PIL algorithm to use for simulation.

The Simulink system path is the full path to the system and “/” must be escaped to “/”. For example, a subsystem named “fuel/sys” inside a model named “demo_fuelsys” has the escaped system path:

```
demo_fuelsys/fuel//sys
```

You obtain the correct system path by clicking the system and then running the `gcb` command. In this example,

```
>> gcb
ans =
demo_fuelsys/fuel//sys
```

- **Configuration** — Allows you to specify a PIL configuration to use for building the PIL application and running the subsequent PIL simulation. For more information, see your Embedded IDE Link documentation.

PIL Block Definitions

PIL Algorithm

The algorithmic code (for example, the control algorithm) that you want to test during the PIL simulation. The PIL algorithm resides in compiled object form to allow verification at the object level.

PIL Application

The executable application to be run on the target platform. The PIL application is created by linking the PIL algorithm object code with some wrapper code (or test harness) that provides an execution framework that interfaces to the PIL algorithm.

The wrapper code includes the `string.h` header file so that the `memcpy` function is available to the PIL application. The PIL application uses `memcpy` to facilitate data exchange between Simulink and the simulation target.

Note The PIL algorithm code under test uses `string.h`. It is independent of the use of `string.h` by the wrapper code. It is entirely dependent on the implementation of the algorithm in the generated code.

Configuring a PIL Simulation

In this section...

“Configuring a Top-Model PIL Simulation” on page 39-14

“Configuring PIL Mode in the Model Block” on page 39-16

“Code Coverage for a PIL Simulation” on page 39-16

Configuring a Top-Model PIL Simulation

To configure and run a top-model PIL simulation for your target environment:

- 1 Open your model.
- 2 Select **Simulation > Processor-in-the-Loop (PIL)**.

Note This option is available only if the model is configured for an ERT or AUTOSAR target. See “Real-Time Workshop Pane: General” and Chapter 22, “Generating Code for AUTOSAR Software Components” for configuration information.

- 3 If you have not already done so, in the Configuration Parameters dialog box, on the **Data Import/Export** pane:

- In the **Input** check box and field, specify stimulus signals (or test vectors) for your top model.
- Configure logging for model outputs, using either *output logging* or *signal logging*:
 - In the **Output** check box and field, specify *output logging*.
 - I/O Support

In the **Signal logging** check box and field, specify *signal logging*.

The software logs only signals that connect to root-level inports and outports. See “Verifying Internal Signals of a Component” on page 36-14.

If the root outputs of your model connect to bus signals, then output logging is not available. Use signal logging for bus signals that connect to root outputs.

You can log signals that connect to inports or outputs of the top model if you name the signals. If you select **Signal logging** but do not name signals that connect to inports or outputs of the top model, then the signal logging object (for example, `logouts`) does not hold signal data.

- Disable logging of Data Store Memory variables. The software does not support this option for this simulation mode. If you do not clear the **Data stores** check box, the software produces a warning when you run the simulation.

For information about the **Data Import/Export** pane, see “Importing and Exporting Data” and “Data Import/Export Pane”.

4 Start the simulation.

Note You cannot:

- Close the model while the simulation is running. To interrupt the simulation, in the Command Window, press **Ctrl+C**.
 - Alter the model during the simulation. You can move blocks and lines as long as it does not alter the behavior of the model.
-

You can run a top-model PIL simulation using the command `sim(model)`.

Note The software supports the `sim` command options `SrcWorkspace` and `DstWorkspace` for the following values only:

- `SrcWorkspace` — 'base'
- `DstWorkspace` — 'base' or 'current'

For more information on the `sim` command and its options, see “Simulation” in the Simulink documentation.

When running a top model in PIL simulation mode, the software ignores the Simulation mode setting for any Model block within the top model. For example, if the top model contains a Model block, the simulation mode of this block may be set to 'Normal', 'Accelerator', 'SIL' or 'PIL', with no effect on the top model PIL simulation.

You control exactly how the code compiles and executes in the target environment through PIL connectivity configurations. See “Creating a Connectivity Configuration for a Target” on page 39-18.

Configuring PIL Mode in the Model Block

To use processor-in-the-loop (PIL) mode in your model, select the Model block simulation mode `Processor-in-the-Loop (PIL)`. This executes the referenced model as compiled code in the target environment.

You control exactly how the code compiles and executes in the target environment through PIL connectivity configurations. See “Creating a Connectivity Configuration for a Target” on page 39-18.

Code Coverage for a PIL Simulation

You can configure code coverage for top-model and Model block SIL simulations. See “Using a Code Coverage Tool in a SIL Simulation” on page 36-22. Code coverage is also available for the corresponding PIL simulations, but only if your PIL application can write directly to the host file system.

For example, you can have a situation where code coverage is not available if you run the PIL application on target hardware. However, you may be able to collect code coverage measurements if you run the PIL application on an instruction set simulator that supports direct file I/O with the host file system.

Creating a Connectivity Configuration for a Target

In this section...

“What Is a PIL Connectivity Configuration?” on page 39-18

“Overview of the Target Connectivity API” on page 39-19

“Creating a Connectivity API Implementation” on page 39-22

“Registering a Connectivity API Implementation” on page 39-22

“Demos of the Target Connectivity API” on page 39-22

What Is a PIL Connectivity Configuration?

You can use PIL connectivity configurations and the target connectivity API to customize PIL to work with any target environment.

Use a connectivity configuration to define:

- A configuration name
- A connectivity API implementation
- Settings that define the set of Simulink models that the configuration is compatible with, for example, the set of models that have a particular system target file, template makefile, and hardware implementation.

You can use the API to integrate third party tools for:

- Building the PIL application, an executable for the target hardware
- Downloading and running the executable
- Communicating with the executable

A particular connectivity configuration name is associated with a single connectivity API implementation. Many different connectivity configurations can coexist and be available for use with PIL simulations. You register each connectivity configuration to Simulink by creating an `sl_customization.m` file and placing it on the MATLAB path.

To run a PIL simulation, the software must first determine which of the available connectivity configurations to use. The software looks for a connectivity configuration that is compatible with the model under test. If the software finds multiple or no compatible connectivity configurations, you see an error message describing how to resolve the problem.

For information on how you create a connectivity configuration for a target, see:

- 1 “Overview of the Target Connectivity API” on page 39-19
- 2 “Creating a Connectivity API Implementation” on page 39-22
- 3 “Registering a Connectivity API Implementation” on page 39-22

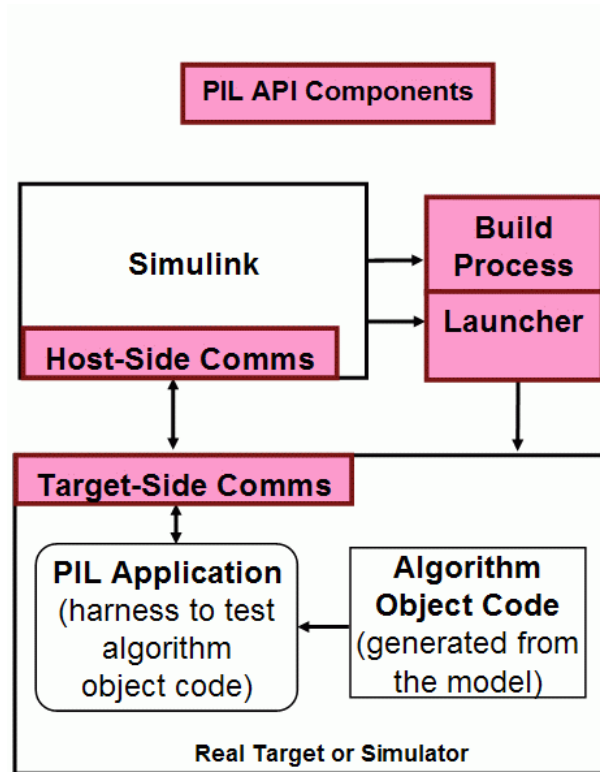
Overview of the Target Connectivity API

- “Target Connectivity API Components” on page 39-19
- “Communications rtiostream API” on page 39-20

Target Connectivity API Components

The following diagram shows what functions the Target Connectivity API components perform:

- Configuring the build process
- Controlling communication between Simulink and the target
- Downloading, starting, and stopping the application on the target



Communications rtiostream API

The communications part of the target connectivity API builds upon the `rtiostream` API, described in this section.

You can use the `rtiostream` API to implement a communication channel to enable exchange of data between different processes. This communication channel is required to enable processor-in-the-loop (PIL) on a new target.

PIL requires a host-target communications channel. This communications channel comprises driver code that runs on the host and target. The `rtiostream` API defines the signature of both target-side and host-side functions that must be implemented by this driver code.

The API is independent of the physical layer that sends the data. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN).

A full `rtiostream` implementation requires both host-side and target-side drivers. Real-Time Workshop software includes host-side drivers for the default TCP/IP implementation (all platforms) as well as a Windows only version for serial communications. To use the TCP/IP `rtiostream` communications channel, you must provide, or obtain from a third party, target-specific TCP/IP device drivers. You must also do this if you require serial communications. For other communication channels and platforms, there is no default implementation provided by Real-Time Workshop software. You must provide both the host-side and the target-side drivers.

The `rtiostream` API comprises the following functions:

- `rtIOStreamOpen`
- `rtIOStreamSend`
- `rtIOStreamRecv`
- `rtIOStreamClose`

You can use `rtiostream_wrapper` to test the `rtiostream` shared library methods from MATLAB code.

To see how the `rtiostream` functions fit into the workflow of creating a connectivity implementation, see the next section, “Creating a Connectivity API Implementation” on page 39-22.

Creating a Connectivity API Implementation

To create a target connectivity API implementation, you must create a subclass of `rtw.connectivity.Config`.

- You must instantiate `rtw.connectivity.MakefileBuilder`. This class configures the build process.
- You must create a subclass of `rtw.connectivity.Launcher`. This class downloads and executes the application using a third-party tool.
- Configure your `rtiostream` communications implementation:
 - On the target-side, integrate the driver code implementing `rtiostream` functions directly into the Real-Time Workshop build process, by creating a subclass of `rtw.pil.RtIOStreamApplicationFramework`.
 - On the host-side, compile the driver code into a shared library. You load and initialize this shared library by instantiating (or optionally, customizing) `rtw.connectivity.RtIOStreamHostCommunicator`.

For all the classes, methods, and functions in the Target Connectivity API, see the following reference section: “Processor-in-the-Loop” in the Real-Time Workshop Embedded Coder Function Reference.

Registering a Connectivity API Implementation

Register the new connectivity API implementation to Simulink as a connectivity configuration, by creating or adding to an `sl_customization.m` file. By doing this, you also define the set of Simulink models that the new connectivity configuration is compatible with.

For details, see `rtw.connectivity.ConfigRegistry` in the Real-Time Workshop Embedded Coder Function Reference.

Demos of the Target Connectivity API

For step-by-step examples, see the following demos:

- `rtwdemo_custom_pil`

This demo shows you how to create a custom PIL implementation using the target connectivity APIs. You can examine the code that configures the

build process to support PIL, a tool to use for downloading and execution, and a communication channel between host and target. Follow the steps to activate a full host-based PIL configuration.

- `rtwdemo_rtiostream`

This demo shows you how to implement a communication channel for use with the Real-Time Workshop Embedded Coder product and your own target support package. This communication channel enables exchange of data between different processes. PIL simulation requires this because it requires exchange of data between the Simulink software running on your host computer and deployed code executing on target hardware.

The `rtiostream` interface provides a generic communication channel that you can implement in the form of target connectivity drivers for a range of connection types. The demo shows how to configure your own target-side driver for TCP/IP, to operate with the default host-side TCP/IP driver. The default TCP/IP communications allow high bandwidth communication between host and target, suitable for transferring data such as video.

The demo also shows how to implement custom target connectivity drivers, for example, using serial, CAN, or USB for both host and target sides of the communication channel.

SIL and PIL Simulation Support and Limitations

In this section...
“SIL and PIL Simulation Support” on page 39-25
“Code Source Support” on page 39-26
“Block Support” on page 39-27
“Configuration Parameters Support” on page 39-29
“I/O Support ” on page 39-33
“Hardware Implementation Support ” on page 39-42
“Other Feature Support” on page 39-46
“PIL Block Limitations” on page 39-46

SIL and PIL Simulation Support

Top-model and Model block software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulation modes are Real-Time Workshop Embedded Coder features.

In addition, you can use top-model and Model block PIL with Embedded IDE Link software specific to AltiumTASKING.

The *PIL block* works with the general Embedded IDE Link product. For more details on the differences between top-model PIL, Model block PIL and the PIL block, see “Choosing a PIL Approach” on page 39-5.

The following tables describe feature support for top-model SIL and PIL, Model block SIL and PIL and the PIL block. “Yes” indicates a supported feature.

This is not a complete list of supported features, but provides information on selected features of interest for SIL and PIL, especially unsupported features and limitations.

Feature	Top-Model PIL	Model Block PIL	PIL Block
Testing of deployment object code	Yes	Yes	Yes
Target Connectivity API	Yes	Yes	No

Code Source Support

Code Source	Code Interface	Top-Model SIL/PIL	Model Block SIL/PIL	PIL Block
Top model	Standalone	Yes	No	Yes
Atomic subsystem	Standalone	No	No	Yes
Virtual subsystem	Standalone	No	No	Yes, but recommend atomic subsystem. See “Algebraic Loop Issues” on page 39-30.
Model block	Model reference Real-Time Workshop target	No, but you can include Model blocks inside your top model.	Yes. See “Cannot Use Multirate Model Block SIL/PIL Inside Conditionally Executed Subsystem” on page 39-27	No, but you can include Model blocks inside your model.
Enabled/ Triggered subsystem	Standalone	No	No	Yes
Export Functions subsystem	Export Functions	No	No	No
Legacy code	Custom	See “Custom Code Interfaces” on page 39-27.	See “Custom Code Interfaces” on page 39-27.	See “Custom Code Interfaces” on page 39-27.
Embedded MATLAB Coder	Embedded MATLAB Coder	See “Custom Code Interfaces” on page 39-27.	See “Custom Code Interfaces” on page 39-27.	See “Custom Code Interfaces” on page 39-27.

For more information on code interfaces, see “SIL Code Interfaces” on page 36-16.

Custom Code Interfaces

The MathWorks does not provide direct PIL support for code interfaces such as legacy code and Embedded MATLAB Coder. However, you can incorporate these interfaces into Simulink as an S-function (for example, using the Legacy Code Tool, S-Function Builder, or hand-written code), and then verify them using PIL.

PIL Does Not Check Real-Time Workshop Error Status

PIL does not check the Real-Time Workshop error status of the generated code under test. This error status flags exceptional conditions during execution of the generated code.

The Real-Time Workshop error status can also be set by blocks in the model (for example, custom blocks developed by a user). It is a limitation that PIL cannot check this error status and report back errors.

Cannot Use Multirate Model Block SIL/PIL Inside Conditionally Executed Subsystem

You see an error if you place your Model block (in either SIL or PIL simulation mode) in a conditionally executed subsystem and the referenced model is multirate (that is, has multiple sample times). Single rate referenced models (with only a single sample time) are not affected.

Block Support

Blocks	Top-Model SIL/PIL	Model Block SIL/PIL	PIL Block
Model block	Yes, you can include Model blocks inside your top model.	Yes	Yes, you can include Model blocks inside your subsystem or model.
Signal Processing Blockset	Yes	Yes	Yes
Video and Image Processing Blockset™	Yes	Yes	Yes

Blocks	Top-Model SIL/PIL	Model Block SIL/PIL	PIL Block
Embedded MATLAB block	Yes	Yes	Yes
Driver blocks	Yes, but not recommended.	Yes, but not recommended.	Yes, but not recommended.
To File blocks	No	No	No
To Workspace blocks	No	No	No
Merge blocks	Yes.	Yes. Cannot connect SIL/PIL outputs to Merge blocks. See “Merge Block Issue” on page 39-28.	Yes. Cannot connect PIL outputs to Merge blocks. See “Merge Block Issue” on page 39-28.
Stop block	No. SIL/PIL ignores the Stop Simulation block and continues simulating.	No. SIL/PIL ignores the Stop Simulation block and continues simulating.	No. PIL ignores the Stop Simulation block and continues simulating.

Run-Time Display Limitation

Scope blocks, and all types of run-time display, such as the display of port values and signal values, have no effect when you specify them in models executing in SIL/PIL mode. The result during simulation is the same as if the constructs did not exist.

Merge Block Issue

If you connect SIL/PIL outputs to a Merge block, you see an error because S-function memory is not reusable.

```
The signal from 'mpil_enabled/Subsystem' output port 1 is
required to be persistent, hence this signal cannot be
connected to a Merge block.
```

Other Top-Model SIL/PIL Limitations

PIL does not support Callbacks (model or block) InitFcn, StartFcn, StopFcn.

Configuration Parameters Support

Configuration Parameters	Top-Model SIL/PIL	Model Block SIL/PIL	PIL Block
ERT-based system target file	Yes	Yes	Yes
AUTOSAR system target file	Yes, but see “AUTOSAR Top Model SIL and PIL Support” on page 22-59	Yes, but see “AUTOSAR Model Block SIL and PIL Support” on page 22-60.	No
GRT-based system target file	No	No	No
GRT compatible call interface	No; see “Missing Code Interface Description File Errors” on page 39-30.	No; see “Missing Code Interface Description File Errors” on page 39-30.	No; see “Missing Code Interface Description File Errors” on page 39-30.
Function Prototype Control	Yes	Yes	Yes
Reusable code format	Yes, but see the special cases in “Imported Data Definitions” on page 39-36.	N/A	Yes, but see the special cases in “Imported Data Definitions” on page 39-36.
Target Function Library	Yes	Yes	Yes
C++	No; see “Missing Code Interface Description File Errors” on page 39-30.	No; see “Missing Code Interface Description File Errors” on page 39-30.	No; see “Missing Code Interface Description File Errors” on page 39-30.
Generate ASAP2 file	Yes	Yes	Yes
Generate example main	N/A	N/A	N/A
MAT-file logging	No	No	No

Configuration Parameters	Top-Model SIL/PIL	Model Block SIL/PIL	PIL Block
Signal logging	Yes, but only for signals connected	No, but see “Verifying Internal Signals of a	No, but see “Verifying Internal Signals of a
’Simplified’ model initialization	Yes	No	Yes
Single output/update	Yes, but see “Algebraic Loop Issues” on page 39-30.	Yes, but see “Algebraic Loop Issues” on page 39-30.	Yes, but see “Algebraic Loop Issues” on page 39-30.
Configuration set reference	Yes	Yes	Depends on the use of the Embedded IDE Link product.

- “Missing Code Interface Description File Errors” on page 39-30
- “Algebraic Loop Issues” on page 39-30

Missing Code Interface Description File Errors

SIL/PIL requires a code interface description file, which is generated during the code generation process for the component under test. If the code interface description file is missing, the SIL/PIL simulation cannot proceed and you see an error reporting that the file does not exist. This error can occur if you select these unsupported options in your configuration parameters:

- **GRT compatible call interface**
- **Target Language option C++ encapsulated**

Make sure that you have not selected these options.

Algebraic Loop Issues

For more information on algebraic loops, see:

- “Algebraic Loops” in the Simulink documentation.
- The Algebraic Loops section in “Simulation Considerations That Affect Code Generation” in the Real-Time Workshop documentation.

- The Introduction section in “Creating Subsystems” in the Real-Time Workshop documentation.

There are three ways that PIL simulation can introduce algebraic loops that do not exist for a normal simulation:

- “Algebraic Loops Caused by Code Generation for a Virtual Subsystem” on page 39-31
- “Algebraic Loops Caused by “Single output/update function”” on page 39-31
- “Algebraic Loops Caused by SIL/PIL Scheduling Limitations” on page 39-31

Algebraic Loops Caused by Code Generation for a Virtual Subsystem.

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 to the simulation behavior.

Declare virtual subsystems as atomic subsystems to ensure consistent simulation and execution behavior for your model.

See “Creating Subsystems” in the Real-Time Workshop documentation.

Algebraic Loops Caused by “Single output/update function”. The “single output/update function” in Real-Time Workshop optimization can introduce algebraic loops because it introduces direct feedthrough via a combined output and update function.

This option is not compatible with the **Minimize algebraic loop occurrences** option (in the Subsystem Parameters dialog box and **Model Referencing** pane of the Configuration Parameters dialog box). This option allows Real-Time Workshop to remove algebraic loops by partitioning generated code appropriately between output and update functions to avoid direct feedthrough.

Algebraic Loops Caused by SIL/PIL Scheduling Limitations. The S-function scheduling mechanism that the software uses to execute the SIL/PIL component has the following limitations:

- Direct feedthrough is always set to true.
- Separate output and update functions in the SIL/PIL component are always executed from the mdlOutputs S-function callback.

These limitations mean that SIL/PIL can introduce algebraic loops that do not exist in normal simulation, and you might get incorrect results. If this happens, you see a warning or error about the introduced algebraic loop and SIL/PIL results may differ from simulation results. You will not see or warning or error if the algebraic loop setting is “none” in the Configuration Parameters dialog box (under Diagnostics on the Solver pane).

A workaround is to break the algebraic loop by inserting a Unit Delay block so that the algebraic loop does not occur. Then, you can use SIL/PIL successfully.

I/O Support

I/O	Top-Model SIL/PIL	Model Block SIL/PIL	PIL Block
Tunable parameters (Model reference arguments)	N/A	Yes, except for tunable structure parameters. See “Tunable Parameters and SIL/PIL” on page 39-36.	N/A
Tunable parameters (Workspace variables)	No	Yes, except for tunable structure parameters. See “Tunable Parameters and SIL/PIL” on page 39-36.	No
Virtual buses	No	Yes	Yes, but some limitations at PIL component boundary; see “PIL Block Virtual Bus Support Limitations” on page 39-40.
Nonvirtual buses	Yes, but see “Top-Model SIL/PIL Bus Limitations” on page 39-40.	Yes	Yes
MUX/DEMUX	No	Yes	Yes, but see “PIL Block MUX Support Limitations” on page 39-41.
Vector/2D/ Multidimensional	Yes	Yes	Yes
Complex data	Yes	Yes	Yes
Fixed-point data	Yes	Yes	Yes

I/O	Top-Model SIL/PIL	Model Block SIL/PIL	PIL Block
Complex fixed-point data	Yes	Yes	Yes
Fixed-point data type override	Not at SIL or PIL component boundary. See “Fixed-Point Tool Data Type Override” on page 39-40	Not at SIL or PIL component boundary. See “Fixed-Point Tool Data Type Override” on page 39-40.	Not at PIL component boundary. See “Fixed-Point Tool Data Type Override” on page 39-40.
Data type replacement	Yes, but see “Data Type Replacement Limitation” on page 39-40	Yes, but see “Data Type Replacement Limitation” on page 39-40	
Goto/From I/O	N/A	N/A	Goto / From blocks must not cross the PIL component boundary. You can use Goto / From blocks to route buried signals up to top-level Inports and Outports <i>inside</i> the PIL component.
Global data store I/O	Yes. See “Global Data Store Support” on page 39-36 and “Imported Data Definitions” on page 39-36.	Yes. See “Global Data Store Support” on page 39-36 and “Imported Data Definitions” on page 39-36.	Yes. See “Global Data Store Support” on page 39-36 and “Imported Data Definitions” on page 39-36.
Local data store I/O	No. See “Imported Data Definitions” on page 39-36.	No. See “Imported Data Definitions” on page 39-36.	No. See “Imported Data Definitions” on page 39-36.
Non-port-based sample times	Yes	Yes	Yes
Continuous sample times	Not at SIL or PIL component boundary.	No	Not at PIL component boundary.

I/O	Top-Model SIL/PIL	Model Block SIL/PIL	PIL Block
Outputs with constant sample time	Yes	No	Yes
Non-auto-storage classes for data (such as signals, parameters, data stores)	Yes. See “Imported Data Definitions” on page 39-36.	Yes. See “Imported Data Definitions” on page 39-36.	Yes. See “Imported Data Definitions” on page 39-36.
Simulink data objects	Yes	Yes	Yes
Simulink numeric type and alias type	Yes	Yes	Yes
Simulink enumerated data	Yes	Yes	Yes
Custom storage classes	Yes, but see “Imported Data Definitions” on page 39-36, and “Unsupported Custom Storage Classes” on page 39-38.	Yes, but see “Imported Data Definitions” on page 39-36, and “Unsupported Custom Storage Classes” on page 39-38.	Yes, but see “Imported Data Definitions” on page 39-36, and “Unsupported Custom Storage Classes” on page 39-38.
Variable-size signals	No. See “Variable-Size Signals and SIL/PIL” on page 39-40.	No. See “Variable-Size Signals and SIL/PIL” on page 39-40.	No. See “Variable-Size Signals and SIL/PIL” on page 39-40.

- “Tunable Parameters and SIL/PIL” on page 39-36
- “Global Data Store Support” on page 39-36
- “Imported Data Definitions” on page 39-36
- “Unsupported Custom Storage Classes” on page 39-38
- “Unsupported Implementation Errors” on page 39-38
- “Variable-Size Signals and SIL/PIL” on page 39-40
- “Fixed-Point Tool Data Type Override” on page 39-40

- “Data Type Replacement Limitation” on page 39-40
- “Top-Model SIL/PIL Bus Limitations” on page 39-40
- “PIL Block Virtual Bus Support Limitations” on page 39-40
- “PIL Block MUX Support Limitations” on page 39-41
- “Incremental Build for Top-Model SIL/PIL” on page 39-41

Tunable Parameters and SIL/PIL

You can tune parameters during a Model block SIL/PIL mode simulation exactly as you do in Normal simulation mode.

For more information, see “Global Tunable Parameters” and “Using Model Arguments” in the Simulink model reference documentation.

Tunable Parameters Limitation. You cannot tune parameters during a SIL/PIL simulation if the parameters have an associated storage class that applies "static" scope or the "const" keyword (for example, Custom, Const, or ConstVolatile). Parameter changes are ignored.

If the storage class also specifies that the parameter is imported, then you must manually define and initialize the value of the parameter. See “Imported Data Definitions” on page 39-36).

Global Data Store Support

SIL/PIL supports global data stores. PIL components that access global data stores must be single rate. You see an error if your SIL/PIL component has multiple sample times and accesses global data stores. To avoid the error, either remove accesses to global data stores or make the component single rate.

Imported Data Definitions

You can use signals, parameters, data stores, etc., that specify storage classes with imported data definitions.

Model Block SIL/PIL. The SIL/PIL application automatically defines storage for imported data associated with:

- Signals at the root level of the component (on the I/O boundary)
- Parameters, except data with "const" type qualifier
- Global data stores

A limitation is that SIL/PIL does not define storage for other imported data storage. You must define the storage through custom code included by the component under test or through the PIL `rtw.pil.RtIOStreamApplicationFramework` API. For example, the PIL application does not define imported data storage for data associated with:

- Internal signals (not on the I/O boundary)
- Local data stores
- Parameters (data with "const" type qualifier)

Top-Model SIL/PIL and PIL Block.

The top-model SIL/PIL or PIL block application automatically defines storage for imported data associated with:

- Signals at the root level of the component (on the I/O boundary)
- Global data stores

A limitation is that SIL/PIL does not define storage for other imported data storage. You must define the storage through custom code included by the component under test or through the PIL `rtw.pil.RtIOStreamApplicationFramework` API. For example, the SIL/PIL application does not define imported data storage for data associated with:

- Internal signals (not on the I/O boundary)
- Local data stores
- Parameters; you must define and initialize parameters

Top-Model SIL/PIL and the PIL block can produce errors if you select the Real-Time Workshop option **Generate reusable code**, and either:

- You have not selected **Inline parameters** and the model contains parameters, or
- You have selected **Inline parameters** and the model contains parameters with SimulinkGlobal storage class.

If either of these conditions are met, then SIL/PIL produces an error similar to the following:

```
Parameter "t_pil_lib_alg/t_pil_lib_alg/Unit Delay:Dialog:X0"  
is not defined in the code associated with the PIL component,  
and is therefore not supported for PIL.  
Please change the parameter's storage class and / or the  
code generation configuration settings so that the parameter  
becomes defined in the code associated with the PIL component.
```

Unsupported Custom Storage Classes

SIL/PIL does not support the following non-addressable custom storage classes:

- BitField
- GetSet

SIL/PIL also does not support signals and parameters with imported grouped custom storage classes.

Unsupported Implementation Errors

You may see errors like the following if you are using a signal or parameter implementation that SIL/PIL does not support:

```
The following data interfaces have  
implementations that are not supported by PIL.
```

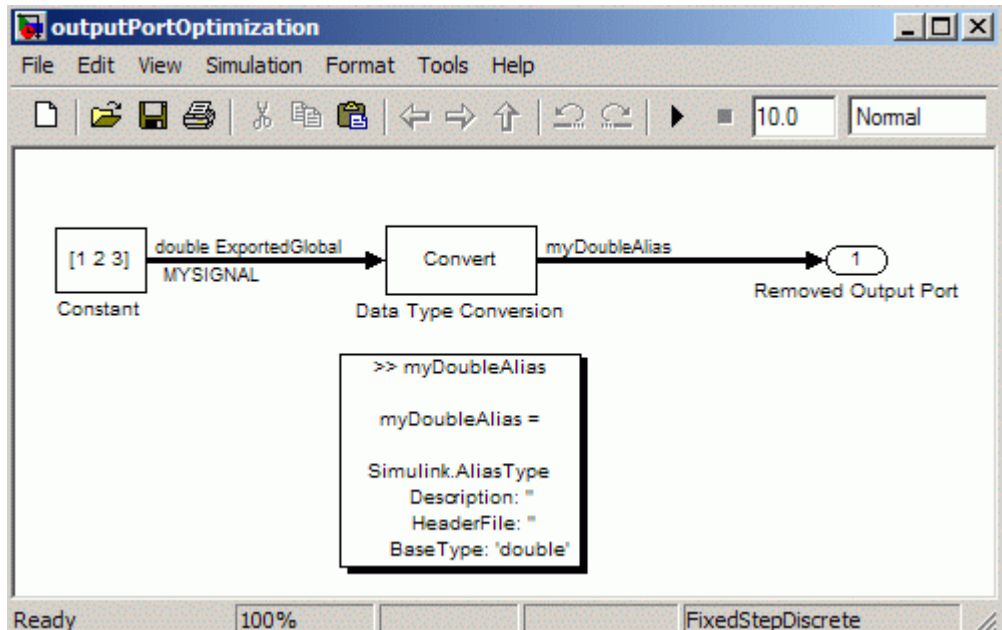
data interfaces may be inports, outputs or parameters.

This error message has the following possible causes:

- The signal or parameter specifies an unsupported custom storage class. See “Unsupported Custom Storage Classes” on page 39-38.

- The model's output port has been optimized through virtual output port optimization. See "Using Virtualized Output Ports Optimization" on page 20-24. The error occurs because the properties (for example, data type, dimensions) of the signal or signals entering the virtual root output port have been modified by routing the signals in one of the following ways:
 - Through a Mux block
 - Through a block that changes the signal's data type. To check the consistency of data types in the model, display Port Data Types by selecting **Format > Port/Signal Displays > Port Data Types**.
 - Through a block that changes the signal dimensions. To check the consistency of data types in the model, display dimensions by selecting **Format > Port/Signal Displays > Signal Dimensions**.

The following example illustrates a model that causes this error due to changing the output port signal's data type.



Variable-Size Signals and SIL/PIL

SIL/PIL treats variable-size signals at the I/O boundary of the SIL/PIL component as fixed-size signals which can lead to errors during propagation of signal sizes. To avoid such errors, use only fixed-size signals at the I/O boundary of the SIL/PIL component.

Fixed-Point Tool Data Type Override

SIL/PIL does not support signals with data types overridden by the Fixed-Point Tool **Data type override** parameter at the SIL/PIL component boundary.

You may see an exception message like the following:

```
Simulink.DataType object 'real_T' is not in scope
from 'mpil_mtrig_no_ic_preread/TmpSFcnForModelReference_unitInTopMdl'.
This error message is related to a hidden S-Function block.
```

There is no resolution for this issue.

Data Type Replacement Limitation

The software does not support replacement data type names that you define for the built-in data type `boolean` if these names map to either the `int` or `uint` built-in data type.

Top-Model SIL/PIL Bus Limitations

The software does not support grounded or unconnected signals at the outputs of a top model.

You must enable the strict bus mode for top-model SIL/PIL:

- 1** In the model window, select **Simulation > Configuration Parameters > Diagnostics > Connectivity**.
- 2** Set **Mux blocks used to create bus signals** to error.

PIL Block Virtual Bus Support Limitations

The PIL block supports virtual buses except for the following cases:

- You see an error if the PIL component is a top model with a root level output that is configured to output a virtual bus. A root level output will output a virtual bus, regardless of the type of the bus that drives it, if it specifies a bus object and the **Output as nonvirtual bus in parent model** check box is not selected.
- You see an error if a right-click subsystem build expands the bus into individual signals.
- For right-click subsystem builds only, the PIL block changes the output of outputs driven by virtual buses (with associated bus objects) into nonvirtual buses. You do not see an error message in this case.

To avoid these limitations, use nonvirtual buses at the PIL component boundary.

PIL Block MUX Support Limitations

The PIL block supports mux signals, except mixed data-type mux signals that expand into individual signals during a right-click subsystem build. You see an error for unsupported cases.

Incremental Build for Top-Model SIL/PIL

When you start a top-model SIL/PIL simulation, the software regenerates code if it detects changes to your model. The software detects changes by using a checksum for the model. However, the software does not detect changes that you make to:

- The `HeaderFile` property of a `Simulink.AliasType` object
- Legacy S-functions

Therefore, if you make these changes, you must build (**Ctrl-B**) your model again before starting the next PIL simulation.

Hardware Implementation Support

Hardware Implementation	Real-Time Workshop Embedded Coder	Embedded IDE Link
Different host and target data-type size	Not at PIL component boundary. See “Hardware Implementation Settings” on page 39-42.	Not at PIL component boundary. See “Hardware Implementation Settings” on page 39-42.
Word-addressable targets	No	Yes
Multiword data type word order different to target byte order	No	Yes
Multiword	No	No
Size of target 'long' > 32 bits	No	No

Hardware Implementation Settings

PIL requires that, in the Simulink Configuration Parameters dialog box, you correctly configure the Hardware Implementation settings for the target environment.

Specify byte ordering for non-8-bit targets.

For more information, see the following sections:

- “Host/Target Data Type Size Mismatch” on page 39-42
- “Data Type Size Mismatch Issues (Real-Time Workshop® Embedded Coder)” on page 39-43
- “Data Type Size Mismatch Issues (Embedded IDE Link)” on page 39-44

Host/Target Data Type Size Mismatch. PIL supports only data types that have the same size on the host and the target at the PIL I/O boundary.

The data types used at the PIL I/O boundary are restricted based on the following rule: PIL supports the data type only if the data-type size on the host (Simulink) is the same as the data-type size on the target.

- For `boolean`, `uint8`, and `int8`, the size is 8-bits.
- For `uint16` and `int16`, the size is 16-bits.
- For `uint32` and `int32`, the size is 32-bits.
- For `single`, the size is 32-bits.
- For `double`, the size is 64-bits.

Examples of unsupported data types are:

- `single` and `double` on targets with 24-bit floating-point types
- `double` on targets with 32-bit double, that is, the same size as `single`

Warning PIL does not always detect unsupported data types (see “Data Type Size Mismatch Issues (Real-Time Workshop® Embedded Coder)” on page 39-43 and “Data Type Size Mismatch Issues (Embedded IDE Link)” on page 39-44). In such cases, data transfer between host and target is incorrect and unexpected data transfer errors occur during the PIL simulation.

To resolve issues with Simulink data types that have different sizes on the host and target, do not use them at the PIL I/O boundary. Instead, use a Simulink data type that maps directly onto a target data type. This resolution is more efficient.

Data Type Size Mismatch Issues (Real-Time Workshop Embedded Coder). PIL mode makes the following assumptions about the target environment:

- The target is byte addressable.
- Sizes of data types on the host and target match.
- Word order of multiword data types on the target is the same as the target byte order.

Warning PIL does not detect violations of these assumptions. If the settings for the target environment violate any of these assumptions, unexpected data transfer errors occur during the PIL simulation.

To resolve issues with Simulink data types that have different sizes on the host and target, do not use them at the PIL I/O boundary. Instead, use a Simulink data type that maps directly onto a target data type. This resolution is more efficient.

Some known violations with predefined hardware implementation settings are:

- Unsupported word addressable targets: TI's C2000™, Freescale™ DSP563xx
- Unsupported data types owing to word order: TASKING Infineon® C166® (single and double have reversed word order)
- Unsupported data types owing to size mismatch: TASKING 8051 (double > is only 4 bytes)

Data Type Size Mismatch Issues (Embedded IDE Link). The Embedded IDE Link product registers the following information about the target environment to Simulink:

- Whether the target is byte addressable.
- The sizes of data types on the target.
- The word order of multiword data types on the target.

This allows PIL to support target environments with unusual hardware characteristics.

PIL can detect data types used at the PIL component boundary that have a host/target data type size mismatch. In such cases, an error indicates that an unsupported data type is being used. This avoids unexpected data transfer errors during simulation.

To resolve issues with Simulink data types that have different sizes on the host and target, do not use them at the PIL I/O boundary. Instead, use a

Simulink data type that maps directly onto a target data type. This resolution is more efficient.

Some target compilers allow the sizes of target data types to be changed from their default size. For example, an IEEE double data type is most likely 8 bytes by default, but an optimization option may be provided to treat it as a 4 byte IEEE single precision type instead. The registration of target data type sizes done by the Embedded IDE Link product is typically statically defined to match the compiler's default data type size, and therefore does not support changing the data type size from that default size.

Warning If you use a nondefault compiler configuration, the target data type sizes registered by the Embedded IDE Link product and the actual target data type sizes may differ. In such cases, data transfer between host and target is incorrect and unexpected data transfer errors occur during the PIL simulation.

To resolve this issue, either:

- Do not use compiler options that change the default size of target data types.
- Use the single data type in Simulink rather than double, if your aim is to treat double-precision floating-point types (8 bytes) as single-precision floating-point types (4 bytes).

Other Feature Support

Other Features	Top-Model SIL/PIL	Model Block SIL/PIL	PIL Block
Multiplatform support (such as Linux)	Yes	Yes	Depends on the use of the Embedded IDE Link product.
Execution profiling	Depends on SIL/PIL configuration.	Depends on SIL/PIL configuration.	Yes (if Embedded IDE Link product support)
Stack profiling	Depends on SIL/PIL configuration.	Depends on SIL/PIL configuration.	Yes (if Embedded IDE Link product support)
C code coverage report	Depends on SIL/PIL configuration.	Depends on SIL/PIL configuration.	Yes (if Embedded IDE Link product support)

PIL Block Limitations

Out of Date PIL Application

The PIL block indicates that the PIL application executable is out of date only when the PIL block detects that new code has been generated for the PIL algorithm under test.

No indication that the PIL application executable is out of date is given under the following circumstances:

- 1 A component of the PIL application executable, such as a C library, is updated after generating code for the PIL algorithm. In this case, click the PIL block **Build** button to bring the PIL application executable up to date. You do not need to regenerate code for the PIL algorithm.
- 2 A model referenced by the PIL algorithm is rebuilt after generating code for the PIL algorithm. In this case, regenerate code for the PIL algorithm and then click the PIL block **Build** button to bring the PIL application executable up to date.

Verifying Numerical Equivalence of Results with Code Generation Verification API

Verifying Numerical Equivalence with Code Generation Verification

In this section...

“Code Generation Verification API Overview” on page 40-2

“Verifying Numerical Equivalence with CGV Workflow” on page 40-2

“Example of Verifying Numerical Equivalence Between Two Modes of Execution of a Model” on page 40-3

“Example of Plotting Output Signals” on page 40-10

Code Generation Verification API Overview

When you execute a model in different modes of execution, you can use the Code Generation Verification (CGV) API to verify the numerical equivalence of results. CGV supports executing the model in simulation, Software-In-the-Loop (SIL), and Processor-In-the-Loop (PIL). For more information about SIL, see Chapter 36, “Verifying Generated Source Code With Software-in-the-Loop Simulation”. For more information about PIL, see Chapter 39, “Verifying Compiled Object Code with Processor-in-the-Loop Simulation”. The CGV demo, `rtwdemo_cgv_script`, shows CGV configuration, execution, and comparison support.

Note CGV helps you verify the numerical equivalence of results for a given set of inputs. CGV can detect numerical deviations for the given set of inputs only. CGV by itself cannot prove the numerical correctness of the generated code. The completeness of the input data that you provide to CGV determines the validity of the results.

Verifying Numerical Equivalence with CGV Workflow

Before verifying numerical equivalence:

- Configure your model for SIL or PIL simulation. For more information, see “Configuring a PIL Simulation” on page 39-14.

- Use the `cgv.Config` class of the CGV API to verify the model configuration for SIL or PIL simulation. For more information, see “Verifying a SIL or PIL Configuration” on page 36-26.
- Configure your model for code generation. For more information, see Preparing Models for Code Generation on page 1.

To verify numerical equivalence:

- Set up the tests for the first execution environment. For example, simulation.
- Use `cgv.CGV.run` to run the tests for the first execution environment.
- Set up the tests for the second execution environment. For example, top-model PIL.
- Use `cgv.CGV.run` to run the tests for the second execution environment.
- Use `cgv.CGV.getOutputData` to get the output data for each execution environment.
- Use `cgv.CGV.getSavedSignals` to display the signal names in the output data. (optional)
- Build a list of signal names for input to other `cgv.CGV` methods. (optional)
- Use `cgv.CGV.createToleranceFile` to create a file method to create a file correlating tolerance information with output signal names. (optional)
- Use `cgv.CGV.compare` to compare the output signals of the first and second execution environments for numerical equivalence.

Example of Verifying Numerical Equivalence Between Two Modes of Execution of a Model

The following example describes configuring, executing, and comparing the results of the `rtwdemo_cgv` model in simulation and SIL modes.

This example contains the following tasks:

- “Configuring the Model” on page 40-4
- “Executing the Model” on page 40-4
- “Comparing All Output Signals” on page 40-6

- “Comparing Individual Output Signals” on page 40-9

Configuring the Model

The first task for verifying numerical equivalence is to check that your model is configured correctly.

- 1 Open the `rtwdemo_cgv` model.

```
cgvModel = 'rtwdemo_cgv';
load_system(cgvModel);
```

- 2 Save the model to a working directory.

```
save_system(cgvModel, fullfile(pwd, cgvModel));
close_system(cgvModel); % avoid original model shadowing saved model
```

- 3 Use the `cgv.Config` class create a `cgv.Config` object to check and modify configuration parameter values, and then save the model for top-model SIL mode of execution.

```
cgvCfg = cgv.Config('rtwdemo_cgv', 'connectivity', 'sil', 'SaveModel', 'on');
```

- 4 Use the `cgv.Config.configModel` method to review your model configuration and change the settings to configure your model for SIL. This method also saves the model because you specify the parameter/value pair ('SaveModel', 'on') when you create the object `cgvCfg`. When 'connectivity' is set to 'sil', the system target file is automatically set to 'ert.tlc'.

```
cgvCfg.configModel(); % Evaluate, change, and save your model for SIL
```

- 5 Display a report of the changes that `cgv.Config.configModel` makes to the model.

```
cgvCfg.displayReport(); % In this example, this reports no changes
```

Executing the Model

Use the CGV API to execute the model in two modes. The two modes in this example are normal mode simulation and SIL mode. In each execution of the model, the CGV object for each mode captures the output data and writes the data to a file.

- 1 If you have not already done so, follow the steps described in “Configuring the Model” on page 40-4.
- 2 Create a `cgv.CGV` object that specifies the `rtwdemo_cgv` model in normal mode simulation.

```
cgvSim = cgv.CGV(cgvModel, 'connectivity', 'sim');
```

Note When the top model is set to Normal simulation mode, any referenced models set to PIL mode will be changed to Accelerator mode.

- 3 Provide the input file to the `cgvSim` object.

```
cgvSim.addInputData(1, [cgvModel '_data']);
```

- 4 Before execution of the model, specify the MATLAB files to execute or MAT-files to load. This step is optional.

```
cgvSim.addPostLoadFiles({[cgvModel '_init.m']});
```

- 5 Specify a location where the object writes all output data and metadata files for execution. This step is optional.

```
cgvSim.setOutputDir('cgv_output');
```

- 6 Execute the model.

```
result1 = cgvSim.run();

*** handling PostLoad file rtwdemo_cgv_init.m
Start CGV execution of model rtwdemo_cgv, ComponentType topmodel, ...
connectivity sim, InputData rtwdemo_cgv_data.mat
End CGV execution: status completed
```

- 7 Get the output data associated with the input data.

```
outputDataSim = cgvSim.getOutputData(1);
```

- 8 For the next mode of execution, SIL, repeat steps 2–7.

```
cgvSil = cgv.CGV( cgvModel, 'Connectivity', 'sil');
```

```

cgvSil.addInputData(1, [cgvModel '_data']);
cgvSil.addPostLoadFiles({'cgvModel '_init.m'});
cgvSil.setOutputDir('cgv_output');
result2 = cgvSil.run();

```

At the MATLAB command line, the result is:

```

*** handling PostLoad file rtwdemo_cgv_init.m
Start CGV execution of model rtwdemo_cgv, ComponentType topmodel, ...
    connectivity sil, InputData rtwdemo_cgv_data.mat

### Starting Real-Time Workshop build procedure for model: rtwdemo_cgv
### Successful completion of Real-Time Workshop build procedure for ...
    model: rtwdemo_cgv
### Preparing to start SIL simulation ...
### Starting SIL simulation for model: rtwdemo_cgv
### Stopping SIL simulation for model: rtwdemo_cgv
End CGV execution: status completed

```

Comparing All Output Signals

After setting up and running the test, compare the outputs by doing the following:

- 1 If you have not already done so, configure and test the model, as described in “Configuring the Model” on page 40-4 and “Executing the Model” on page 40-4.
- 2 Test that the execution of the model ran successfully:

```

if ~result1 || ~result2
    error('Execution of model failed.');
```

```

end

```

- 3 Use the `cgv.CGV.getOutputData` method to get the output data from the `cgv.CGV` objects.

```

simData = cgvSim.getOutputData(1);
silData = cgvSil.getOutputData(1);

```

- 4 Display a list of signals by name using the `cgv.CGV.getSavedSignals` method.

```
cgvSim.getSavedSignals(simData);
```

At the MATLAB command line, the result is:

```
simData.hi0.Data(:,1)
simData.hi0.Data(:,2)
simData.Vector.Data(:,1)
simData.Vector.Data(:,2)
simData.Vector.Data(:,3)
simData.Vector.Data(:,4)
simData.BusOutputs.hi0.Data(:,1)
simData.BusOutputs.hi0.Data(:,2)
simData.BusOutputs.hi1.mid0.lo0.Data(1,1,:)
simData.BusOutputs.hi1.mid0.lo0.Data(1,2,:)
simData.BusOutputs.hi1.mid0.lo0.Data(2,1,:)
simData.BusOutputs.hi1.mid0.lo0.Data(2,2,:)
simData.BusOutputs.hi1.mid0.lo1.Data
simData.BusOutputs.hi1.mid0.lo2.Data
simData.BusOutputs.hi1.mid1.Data(:,1)
simData.BusOutputs.hi1.mid1.Data(:,2)
simData.ErrorsInjected.Data
```

- 5** Using the list of signals, build a list of signals in a cell array of strings. The signal list can contain any number of signals.

```
signalList = {'simData.ErrorsInjected.Data'};
```

- 6** Use the `cgv.CGV.createToleranceFile` method to create a file, in this example, 'localtol', correlating tolerance information with output signal names.

```
toleranceList = {'absolute', 0.5};
cgv.CGV.createToleranceFile('localtol', signalList, toleranceList);
```

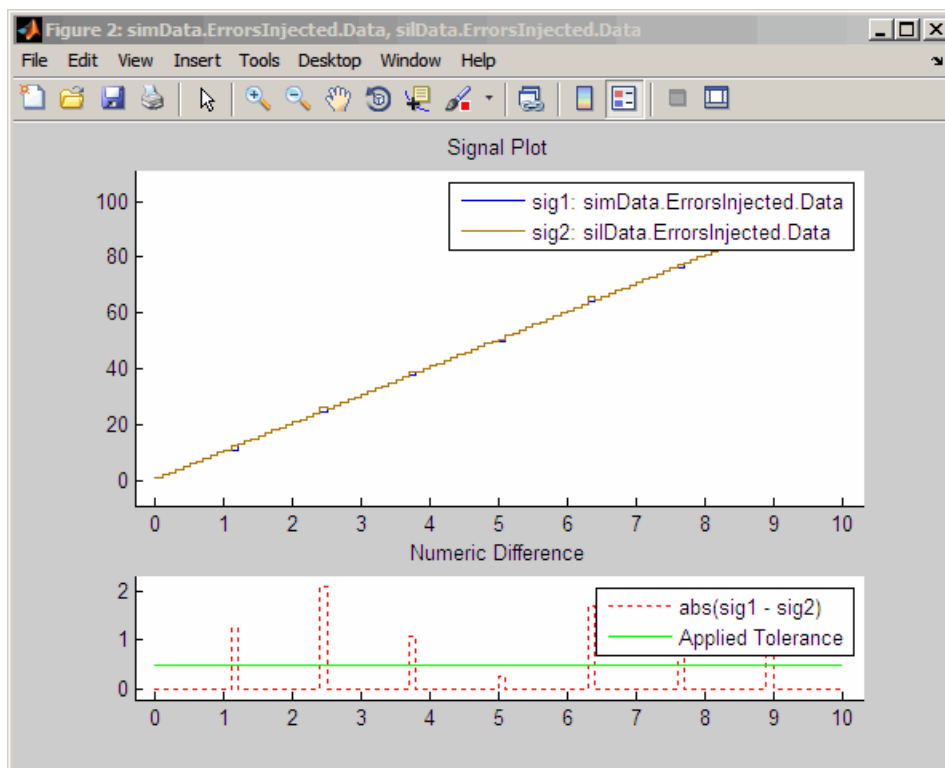
- 7** Compare the output data signals. By default, the `cgv.CGV.compare` method looks at all signals which have a common name between both executions. If a tolerance file is present, `cgv.CGV.compare` uses the associated tolerance for a specific signal during comparison; otherwise the tolerance is zero. In this example, the 'Plot' parameter is set to 'mismatch'. Therefore, only mismatched signals produce a plot.

```
[matchNames, ~, mismatchNames, ~] = ...
    cgV.CGV.compare(simData, silData, 'Plot', 'mismatch', ...
        'Tolerancefile', 'localtol');
fprintf( '%d Signals match, %d Signals mismatch\n', ...
    length(matchNames), length(mismatchNames));
disp('Mismatched Signal Names:');
disp(mismatchNames);
```

At the MATLAB command line, the result is:

```
14 Signals match, 1 Signals mismatch
Mismatched Signal Names:
    'simData.ErrorsInjected.Data'
```

A plot results from the mismatch on signal `simData.ErrorsInjected.Data`.



The lower plot displays the numeric difference between the results.

Comparing Individual Output Signals

After setting up and running the test, compare the outputs of individual signals by doing the following:

- 1 If you have not already done so, configure and test the model, as described in “Configuring the Model” on page 40-4 and “Executing the Model” on page 40-4.
- 2 Use the `cgv.CGV.getOutputData` method to get the output data from the `cgv.CGV` objects.

```
simData = cgvSim.getOutputData(1);
silData = cgvSil.getOutputData(1);
```

- 3 Use the `cgv.CGV.getSavedSignals` method to display the output data signal names. Build a list of specific signal names in a cell array of strings. The signal list can contain any number of signals.

```
cgv.CGV.getSavedSignals(simData);
signalList = {'simData.BusOutputs.hi1.mid0.lo1.Data', ...
'simData.BusOutputs.hi1.mid0.lo2.Data', 'simData.Vector.Data(:,3)'};
```

- 4 Use the specified signals as input to the `cgv.CGV.compare` method to compare the signals from separate runs.

```
[matchNames, ~, mismatchNames, ~] = ...
    cgv.CGV.compare(simData, silData, 'Plot', 'mismatch', ...
    'signals', signalList);
fprintf( '%d Signals match, %d Signals mismatch\n', ...
    length(matchNames), length(mismatchNames));
if ~isempty(mismatchNames)
    disp( 'Mismatched Signal Names:');
    disp(mismatchNames);
end
```

At the MATLAB command line, the result is:

```
3 Signals match, 0 Signals mismatch
```

Example of Plotting Output Signals

After setting up and running the test, use the `cgv.CGV.plot` method to plot output signals.

- 1** If you have not already done so, configure and test the model, as described in “Configuring the Model” on page 40-4 and “Executing the Model” on page 40-4.
- 2** Use the `cgv.CGV.getOutputData` method to get the output data from the `cgv.CGV` objects.

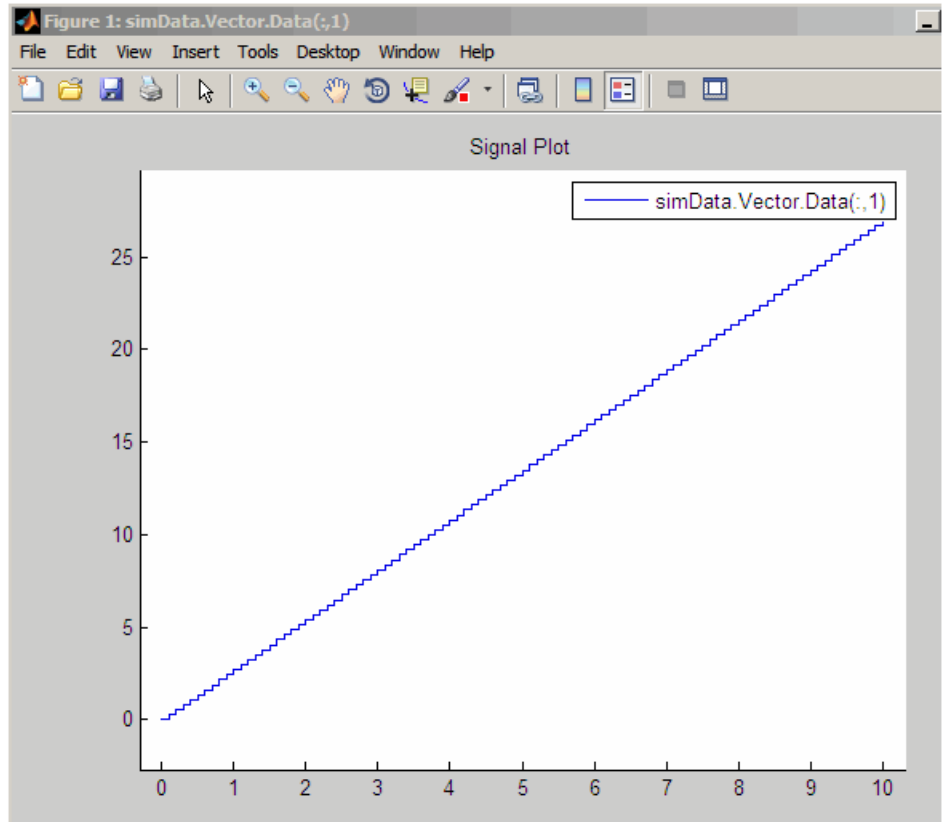
```
simData = cgvSim.getOutputData(1);
```

- 3** Use the `cgv.CGV.getSavedSignals` method to display the output data signal names. Build a list of specific signal names in a cell array of strings. The signal list can contain any number of signals.

```
cgv.CGV.getSavedSignals(simData);  
signalList = {'simData.Vector.Data(:,1)'};
```

- 4** Use the specified signal list as input to the `cgv.CGV.plot` method to compare the signals from separate runs.

```
[signalNames, signalFigures] = cgv.CGV.plot(simData, ...  
    'Signals', {'simData.Vector.Data(:,1)'});
```



Examples

Use this list to find examples in the documentation.

Code Generation

- “Generating a Shared Library Version of Your Model Code” on page 3-10
- “Creating Application Code to Load and Use Your Shared Library File” on page 3-11
- “Specifying Code Generation Objectives Using the GUI” on page 14-4
- “Specifying Code Generation Objectives at the Command Line” on page 14-6
- “Reviewing the Model Without Generating Code” on page 14-7
- “Reviewing the Model During Code Generation” on page 14-9
- “How to Create Custom Objectives” on page 14-14
- “Using Virtualized Output Ports Optimization” on page 20-24
- “Importing an AUTOSAR Software Component” on page 22-30
- “Creating a SIL Block” on page 24-4
- “Techniques for Exporting Function-Call Subsystems” on page 25-7
- “Function-Call Subsystem Export Example” on page 25-10
- “Examples of Modular Function Code for Nonvirtual Subsystems” on page 26-9
- “Model Function Prototypes Example” on page 27-12
- “Sample Script for Configuring Model Function Prototypes” on page 27-22
- “C++ Encapsulation Quick-Start Example” on page 28-4
- “Sample Script for Configuring the Step Method for a Model Class” on page 28-25

Custom Storage Classes

- “Creating Packages that Support CSC Definitions” on page 7-8
- “Generating Code with Custom Storage Classes” on page 7-58
- “Defining Advanced Custom Storage Class Types” on page 7-62
- “GetSet Custom Storage Class Example” on page 7-68
- “Assigning a Custom Storage Class to Data” on page 7-79

Memory Sections

- “Requirements for Defining Memory Sections” on page 8-4

- “Defining Memory Sections” on page 8-7
- “Applying Memory Sections” on page 8-12
- “Examples of Generated Code with Memory Sections” on page 8-20

Advanced Code Generation

- “Specifying Type Definition Location for User-Defined Data Types” on page 10-5
- “Generating Source and Header Files with a Custom File Processing (CFP) Template” on page 16-34
- “Creating a Custom File and Function Banner Template” on page 16-52
- “Customizing a Code Generation Template (CGT) File for File and Function Banner Generation” on page 16-53
- “Adding a Configuration Wizard Block to Your Model” on page 20-8
- “Creating a Custom Configuration Wizard Block” on page 20-11

Target Function Libraries

- “Target Function Libraries Quick-Start Example” on page 29-9
- “Example: Mapping Math Functions to Target-Specific Implementations” on page 29-27
- “Example: Mapping the memcpy Function to a Target-Specific Implementation” on page 29-34
- “Example: Mapping Nonfinite Support Utility Functions to Target-Specific Implementations” on page 29-39
- “Example: Mapping Scalar Operators to Target-Specific Implementations” on page 29-44
- “Adding Target Function Library Reserved Identifiers” on page 29-137
- “Examining and Validating Function Replacement Tables” on page 29-140
- “Using the sl_customization API to Register a TFL with Simulink Software” on page 29-150
- “Registering Multiple TFLs” on page 29-156

Data Structures, Code Modules, and Program Execution

“Rate Grouping and the Static Main Program” on page 31-17

“Making Your S-Functions Rate Grouping Compliant” on page 31-21

Verifying Generated Code

“Configuring a Top-Model PIL Simulation” on page 39-14

“Configuring PIL Mode in the Model Block” on page 39-16

“Demos of the Target Connectivity API” on page 39-22

A

- absolute time 30-3
- additional options
 - adding custom comments 16-4
 - delimiter for all `#includes` 16-68
- algorithms
 - verifying in context of complete real-time target environment 38-1
- Alias 6-10
- attributes 11-3
- AUTOSAR 21-2
 - standards, complying with 21-5

B

- Bitfield (Custom) 6-7
- Browse button
 - on Real-Time Workshop pane 15-2

C

- changing identifier names 11-30
- classes 11-3
- code generation options
 - Application lifespan (days) 20-3
 - Bitfield declarator type specifier 20-2
 - code style pane 16-21
 - Configure Step Function 30-6
 - Custom comments 16-2
 - External mode 20-26
 - Fixed-point exception protection 20-3
 - Generate reusable code 30-5
 - Generate scalar inlined parameters 16-11
 - GRT compatible call interface 30-5
 - Identifier format control 16-11
 - MAT-file logging
 - clearing 20-23
 - Maximum identifier length 16-11
 - Minimum mangle length 16-11
 - Pack Boolean data into bitfields 20-2

- Parameter structure 20-2
- Pass reusable subsystem outputs as 20-2
- Pass root-level I/O as 30-7
- Requirements in block comments 16-3
- Reusable code error diagnostic 30-7
- Simplify array indexing 20-2
- Simulink block descriptions 16-2
- Simulink data object descriptions 16-2
- Single output/update function 30-5
 - clearing 31-19
- Stateflow object descriptions 16-3
- Support absolute time 30-3
- Support complex numbers 30-2
- Support continuous time 30-3
 - for using continuous time blocks 4-2
 - limitations 24-6
- Support floating-point numbers 30-2
- Support non-finite numbers 30-2
- Support noninlined S-functions 30-4
- Suppress error status in real-time model data structure 30-6
 - Terminate function required 30-5
- Code generation options 6-4
- code modules, generated 18-2
- code style
 - controlling 16-21
- code templates
 - example of use 16-34
 - generating code with 16-35
 - structure of 16-30
 - summary of API 16-46
- code tracing
 - by using HTML reports 34-2
- code, generated
 - verifying in target environment 37-1
- code, user-written 18-5
- comments
 - adding custom 16-4
 - adding global 16-5
- Complexity 6-3

- component models
 - verifying in context of complete real-time target environment 38-1
 - Configuration Parameters dialog box 17-3
 - Configuration Wizard buttons 20-6
 - Const (Custom) 6-7
 - ConstVolatile (Custom) 6-8
 - controllers
 - verifying in context of complete real-time target environment 38-1
 - controlling signal storage 20-25
 - creating a data dictionary 11-4
 - custom code generation
 - of file banners 16-49
 - with code templates 16-30
 - custom comments 16-4
 - custom file processing (CFP) template 16-22
 - custom storage classes
 - assigning to data 7-79
 - code generation with 7-80
 - instance-specific attributes 7-78
- D**
- daexplr command 11-8
 - Data access 6-9
 - data dictionary 11-2
 - introduction 11-2
 - See also* data objects
 - data initialization
 - of floats and doubles 20-2
 - of internal states 20-2
 - of root-level I/O ports 20-2
 - data object wizard 11-5
 - data objects
 - adding missing 11-5
 - naming rules
 - changing all #defines 11-36
 - changing all parameter names 11-35
 - changing all signal names 11-35
 - properties 6-2
 - setting property values 11-8
 - wizard 11-5
 - data placement
 - introduction 12-2
 - rules for 12-25
 - settings 12-2
 - data templates 16-22
 - Data type property 6-3
 - data types
 - creating 11-38
 - dataobjectwizard 11-6
 - Default (Custom) storage class 6-5
 - Define (Custom) 6-8
 - #defines
 - changing all 11-36
 - defining all objects in separate file 11-26
 - defining one object in its own file 11-28
 - Definition file 6-6
 - Definition File priority 12-9
 - Description 6-10
 - design specification
 - developing 2-1 3-1 4-1
 - designs, model
 - optimizing for specific hardware with on-target rapid prototyping 35-1
 - dialog boxes
 - AUTOSAR Options 22-32
 - Model Interface 27-4
 - Dialog boxes
 - Model Explorer 11-8
 - Dimensions 6-3
 - Direct 6-9
 - DO-178B 21-2
 - standards, complying with 21-8
 - DO-178B Standard 21-8
 - DO178-B 21-2
 - DocBlock 16-6
 - domain
 - installing products for 2-1

E

- elapsed time 30-3
- entry points, model 31-14
- ert_main.c 31-16
- ert_main.cpp 31-16
- ExportToFile (Custom) 6-8
- external data dictionary
 - importing data objects from 11-16
- External mode support 20-26

F

- file banners, generation of 16-49
- file packaging 18-2
- Frame based 6-4

G

- generated code
 - modules 18-2
 - verifying in target environment 37-1
- Get function 6-9
- GetSet (Custom) 6-9
- GetSet custom storage class 7-66
- Global (Custom) storage class 6-5
- global comments
 - using DocBlock 16-6
 - using Simulink annotation 16-7
 - using sorted notes 16-9
 - using Stateflow note 16-8
- Global priority 12-7
- guidelines
 - MISRA C® 21-5

H

- Hardware Implementation parameters
 - configuration of 20-20
- Header file 6-6
- Header File priority 12-9

I

- identifier format control parameters 16-14
- identifier format control tokens 16-13
- IEC 61508 21-2
 - standards, complying with 21-6
- IEC 61508 Standard 21-6
- ImportFromFile (Custom) 6-9
- #include
 - specifying delimiter 16-68
- industry standards and guidelines
 - modeling and coding 21-2
- inserting custom comments 16-4
- inserting global comments 16-5
- integer-only code 20-23
- integer-only code generation 20-23
- interrupts, servicing 31-4

M

- main program (ert_main)
 - modifying 31-5
 - operation of 31-5
 - static module 31-16
 - VxWorks example 32-1
- MathWorks Automotive Advisory Board (MAAB)
 - guidelines, complying with 21-4
- MathWorks™ Automotive Advisory Board (MAAB) 21-2
- MATLAB functions
 - #define naming 11-36
 - parameter naming 11-35
 - signal naming 11-35
- Maximum property 6-4
- MemConst 6-5
- MemConstVolatile 6-5
- Memory section 6-5
- MemVolatile 6-5
- Minimum property 6-4
- MISRA C 21-2
- MISRA C®

- guidelines, complying with 21-5
- MISRA C[®] guidelines 21-5
- model design specification
 - developing 2-1 3-1 4-1
- model designs
 - optimizing for specific hardware with on-target rapid prototyping 35-1
- model entry points 31-14
- Model Explorer
 - parameter and signal properties 6-2
- modifying `rt_OneStep` 31-12
- mpt (module packaging tool) data object 11-4

N

- name mangling 16-15
- naming rules
 - applying globally 11-30
 - changing all `#defines` 11-36
 - changing all parameter names 11-35
 - changing all signal names 11-35

O

- on-target rapid prototyping
 - optimizing model designs for specific hardware with 35-1
- Owner 6-6
- ownership
 - effects of settings 12-10
 - explanation 12-10
- Ownership priority 12-9

P

- package 11-3
- Parameter class 11-3
- parameter names
 - changing all 11-35
- Persistence level 6-7
- PIL

- introduction 39-1
- PIL simulation
 - definitions 39-13
- Pointer 6-9
- priority and usage 12-3
 - Definition File priority 12-9
 - Global priority 12-7
 - Header File priority 12-9
 - introduction 12-3
 - Ownership priority 12-9
 - Read-Write priority 12-5
 - See also* interdependent settings
- processor-in-the-loop (PIL)
 - communications API 39-20
 - connectivity API 39-8
 - connectivity API demos 39-22
 - connectivity configuration 39-18
 - custom target 39-18
 - Embedded IDE Link[™] product 39-10
 - how PIL works 39-3
 - introduction 39-1
 - limitations 39-24
 - Model block simulation mode 39-16
 - rtiostream API 39-20
 - target connectivity API 39-18
 - top-model simulation mode 39-14
- program execution
 - main program 31-5
 - `rt_OneStep` 31-7
 - multirate multitasking operation 31-9
 - multirate single-tasking operation 31-12
 - reentrancy 31-11
 - single-rate single-tasking operation 31-8
- property values
 - definition 11-2
 - descriptions 6-2
 - setting 11-8
- pure integer code 20-23
 - and external mode 20-27

R

- rapid prototyping
 - optimizing model designs for specific hardware with 35-1
- rate grouping 31-10
- Read-Write priority 12-5
- Real-Time Workshop pane
 - target configuration options
 - Browse button 15-2
 - system target file field 15-2
- Real-Time Workshop Report 11-25
- reentrant code 30-5
- requirements
 - validating with traceability 33-1
- reusable code 30-5
- RTW.copyFileToBuildDir 29-135
- rtwdemo_mpf.mdl 11-19

S

- S-function wrapper generation 24-1
- Sample based 6-4
- Sample mode 6-4
- Sample time 6-3
- Set function 6-9
- Signal class 11-3
- signal names
 - changing all 11-35
- Simulink annotation 16-7
- solver modes, permitted 31-7
- sorted notes 16-9
- source code files, generated 18-2
- specification
 - developing 2-1 3-1 4-1
- standards
 - DO-178B 21-8
 - IEC 61508 21-6
- standards and guidelines, modeling and coding 21-2
- Stateflow note 16-8

- Storage class 6-4
- Struct (Custom) 6-9
- Struct name 6-7
- symbols for templates
 - alphabetical list 16-61
- System Target File Browser 15-5
- system target files
 - selecting programmatically 15-7

T

- target connectivity API 39-18
- target environment
 - verifying generated code in 37-1
- target function library. *See* TFL
- targets
 - selecting programmatically 15-7
- task identifier (tid) 31-9
- templates
 - example with generated file 16-43
 - rules for creating or modifying 16-65
 - symbols 16-61
- TFL 29-1
 - build information for function
 - replacements 29-134
 - cache hits and misses 29-145
 - conceptual view of function or operator 29-3
 - defining 29-16
 - examining 29-140
 - registering 29-149
 - reserved identifiers 29-137
 - RTW.copyFileToBuildDir 29-135
 - rtwTargetInfo.m file 29-149
 - selecting in MATLAB Embedded Coder 29-8
 - selecting in Simulink 29-8
 - sl_customization.m file 29-149
 - table definition file 29-16
 - table entry 29-3
 - target-specific implementation of function or operator 29-3

- TargetFcnLibHandle 29-145
- tracing generated code 29-144
- validating 29-140
- Viewer 29-141
- workflow 29-7
- tid 31-9
- timer interrupts 31-4
- traceability
 - for validating requirements 33-1
- tutorial
 - changing identifier names 11-31
 - changing organization of generated file 16-32
 - creating a data dictionary 11-19
 - defining all objects in separate file 11-26
 - defining one object in its own file 11-28

U

- Units 6-3
- User data type 11-38
- User object type 6-2

V

- Value 6-3
- virtualized output port optimization 20-24
- Volatile (Custom) 6-7
- VxWorks deployment example 32-1

W

- wizard
 - data object 11-5