

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 2002–2008 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
December 2003 Online only
June 2004 Online only
October 2004 Online only
March 2005 Online only
September 2005 Online only
March 2006 Online only
September 2006 Online only
March 2007 Online only
September 2007 Online only
March 2008 Online only
October 2008 Online only

Version 3.0 (Release 13)
Revised for Version 3.2 (Release 13SP1+)
Revised for Version 4.0 (Release 14)
Revised for Version 4.1 (Release 14SP1)
Revised for Version 4.2 (Release 14SP2)
Revised for Version 4.3 (Release 14SP3)
Revised for Version 4.4 (Release 2006a)
Revised for Version 4.5 (Release 2006b)
Revised for Version 4.6 (Release 2007a)
Revised for Version 5.0 (Release 2007b)
Revised for Version 5.1 (Release 2008a)
Revised for Version 5.2 (Release 2008b)

Introduction to the Real-Time Workshop® Embedded Coder Product

1

Data Structures, Code Modules, and Program Execution

2

Real-Time Model (rtModel) Data Structure	2-2
Overview	2-2
rtModel Accessor Macros	2-3
Code Modules	2-4
Introduction	2-4
Generated Code Modules	2-4
User-Written Code Modules	2-7
Generating the Main Program Module	2-8
Program Execution Overview	2-10
Stand-Alone Program Execution	2-11
Overview	2-11
Main Program	2-12
rt_OneStep	2-14
Wind River Systems VxWorks Example Main Program Execution	2-21
Introduction to the VxWorks Example Main Program	2-21
Task Management	2-22
Model Entry Points	2-24

Static Main Program Module	2-26
Overview	2-26
Rate Grouping and the Static Main Program	2-27
Modifying the Static Main Program	2-28
Rate Grouping Compliance and Compatibility	
Issues	2-31
Main Program Compatibility	2-31
Making Your S-Functions Rate Grouping Compliant	2-31

Code Generation Options and Optimizations

3

Accessing the ERT Target Options	3-2
Introduction	3-2
Viewing ERT Target Options in the Configuration Parameters Dialog Box or Model Explorer	3-3
Support for Continuous Time Blocks, Continuous Solvers, and Stop Time	3-4
Generating Code for Continuous Time Blocks	3-4
Generating Code that Supports Continuous Solvers	3-4
Generating Code that Honors a Stop Time	3-5
Mapping Application Requirements to Configuration Options	3-6
Configuring a Model	3-7
Selecting an ERT Target	3-7
Generating a Report that Includes Hyperlinks for Tracing Code to Model Blocks	3-9
Customizing Comments in Generated Code	3-10
Customizing Generated Identifiers	3-12
Configuring Symbols	3-13
Configuring Model Interfaces	3-21
Controlling Code Style	3-26
Configuring Templates for Customizing Code	3-27
Configuring the Placement of Data in Code	3-28
Configuring Replacement Data Types	3-28

Configuring Memory Sections	3-30
Configuring Optimizations	3-31
Tips for Optimizing the Generated Code	3-33
Introduction	3-33
Using Auto-Optimized Targets	3-33
Using Configuration Wizard Blocks	3-34
Setting Hardware Implementation Parameters	
Correctly	3-34
Removing Unnecessary Initialization Code	3-36
Generating Pure Integer Code If Possible	3-37
Disabling MAT-File Logging	3-37
Using Virtualized Output Ports Optimization	3-38
Using Stack Space Allocation Options	3-39
Using External Mode with the ERT Target	3-41
Creating and Using a Code Generation Report	3-43
Overview	3-43
Generating an HTML Code Generation Report	3-44
Using Code-to-Model Traceability	3-46
Using Model-to-Code Traceability	3-48
Using the Model-to-Code Navigation Dialog Box to Load	
Existing Trace Information	3-50
Using the Code Interface Report to Analyze the Generated	
Code Interface	3-51
Using the Traceability Report to Completely Map Model	
Elements to Code	3-63
HTML Code Generation Report Limitations	3-65
Automatic S-Function Wrapper Generation	3-68
Overview	3-68
Generating an ERT S-Function Wrapper	3-69
S-Function Wrapper Generation Limitations	3-71
Exporting Function-Call Subsystems	3-72
Overview	3-72
Exported Subsystems Demo	3-73
Additional Information	3-73
Requirements for Exporting Function-Call Subsystems ..	3-73
Techniques for Exporting Function-Call Subsystems	3-75
Optimizing Exported Function-Call Subsystems	3-76

Exporting Function-Call Subsystems That Depend on Elapsed Time	3-76
Function-Call Subsystem Export Example	3-77
Function-Call Subsystems Export Limitations	3-81
Nonvirtual Subsystem Modular Function Code	
Generation	3-83
Overview	3-83
Configuring Nonvirtual Subsystems for Generating Modular Function Code	3-84
Examples of Modular Function Code for Nonvirtual Subsystems	3-88
Nonvirtual Subsystem Modular Function Code Limitations	3-94
Controlling Model Function Prototypes	
Overview	3-96
Configuring Model Function Prototypes	3-97
Model Function Prototypes Example	3-105
Configuring Model Function Prototypes Programmatically	3-111
Sample M Script for Configuring Model Function Prototypes	3-114
Verifying Generated Code for Customized Functions	3-115
Model Function Prototype Control Limitations	3-115
Generating and Controlling C++ Encapsulation	
Interfaces	3-118
Overview of C++ Encapsulation	3-118
C++ Encapsulation Quick-Start Example	3-119
Generating and Configuring C++ Encapsulation Interfaces to Model Code	3-126
Configuring C++ Encapsulation Interfaces Programmatically	3-133
Sample M-Script for Configuring the Step Method for a Model Class	3-136
C++ Encapsulation Interface Control Limitations	3-138
Creating and Using Host-Based Shared Libraries	
Overview	3-140
Generating a Shared Library Version of Your Model Code	3-141

Creating Application Code to Load and Use Your Shared Library File	3-142
Host-Based Shared Library Limitations	3-146

Verifying Generated Code

4

Verifying Generated Code with	
Software-in-the-Loop	4-2
Overview	4-2
Validating Generated Code on the MATLAB Host Computer	
Using Hardware Emulation	4-3
Validating ERT Production Code on the MATLAB Host	
Computer Using Portable Word Sizes	4-4
Verifying Generated Code with	
Processor-in-the-Loop	4-7
What Is Processor-in-the-Loop Cosimulation?	4-7
Comparison of PIL and SIL	4-7
Why Use PIL Cosimulation?	4-8
Problems That PIL Can Detect	4-8
How PIL Cosimulation Works	4-9
Types of PIL Functionality	4-10
PIL Mode in the Model Block	4-10
PIL Behavior and Model Referencing	4-11
PIL Block For Embedded IDE Link Products	4-12
Code Interfaces For PIL	4-13
Comparison of the Model Block in PIL Mode and the PIL Block	4-15
PIL Modeling Scenarios with the Model Block	4-15
Modeling Scenarios with the PIL Block	4-20
Configuring PIL Mode in the Model Block	4-22
Using PIL Mode in the Model Block	4-22
Using the Default PIL Connectivity Configuration	4-22
PIL Connectivity Configurations	4-24

Creating a Connectivity Configuration for Your Target	4-26
Overview of the Target Connectivity API	4-26
Create Your Connectivity API Implementation	4-29
Register Your Connectivity API Implementation	4-29
Demos of the Target Connectivity API	4-29
PIL Feature Support and Limitations	4-31
PIL Functionality Support	4-32
Code Source Support	4-33
Block Support	4-34
Configuration Parameters Support	4-36
Signal Data Type and I/O Support	4-40
Hardware Implementation Support	4-42
Other Feature Support	4-47

Developing Models and Code That Comply with Industry Standards and Guidelines

5

What Are the Standards and Guidelines?	5-2
Developing Models and Code That Comply with MAAB Guidelines	5-4
Developing Models and Code That Comply with MISRA C Guidelines	5-5
Generating Code That Complies with AUTOSAR Standards	5-6
Overview of AUTOSAR Support	5-6
Importing an AUTOSAR Software Component	5-8
Using the Configure AUTOSAR Interface Dialog Box	5-10
Configuring Ports for Basic Software and Error Status Receivers	5-14
Configuring Calibration Parameters	5-15
Modifying and Validating an Existing AUTOSAR Interface	5-16

Exporting a Single Runnable Entity AUTOSAR Software Component	5-17
Configuring Multiple Runnable Entities	5-19
Configuring AUTOSAR Options Programmatically	5-21
Verifying the AUTOSAR Code Using Software-in-the-Loop Testing	5-21
Limitations and Tips	5-22
Migrating AUTOSAR Development Kit Models	5-23
Demos and Further Reading	5-23

Developing Models and Code That Comply with the IEC 61508 Standard	5-25
Applying Simulink and Real-Time Workshop Technology to the IEC 61508 Standard	5-25
Checking for IEC 61508 Standard Compliance Using the Model Advisor	5-25
Validating Traceability	5-26

Developing Models and Code That Comply with the DO-178B Standard	5-28
Applying Simulink and Real-Time Workshop Technology to the DO-178B Standard	5-28
Checking for Standard Compliance Using the Model Advisor	5-28
Validating Traceability	5-29

Custom Storage Classes

6

Introduction to Custom Storage Classes	6-2
Custom Storage Class Memory Sections	6-5
Predefined Custom Storage Classes	6-6
Changing Custom Storage Classes	6-8
Setting Custom Storage Class Properties	6-9

Generating Code with CSCs	6-11
Set Model Properties	6-11
Instantiate Signal Objects	6-12
Creating Packages with CSC Definitions	6-16
Designing Custom Storage Classes and Memory	
Sections	6-20
Using the Custom Storage Class Designer	6-20
Editing Custom Storage Class Properties	6-25
Using Custom Storage Class References	6-32
Editing Memory Section Definitions	6-35
Using Memory Section References	6-38
Defining Advanced Custom Storage Class Types	6-42
Overview	6-42
Create Your Own Parameter and Signal Classes	6-42
Create a Custom Attributes Class for Your CSC (Optional)	6-43
Write TLC Code for Your CSC	6-43
Register Custom Storage Class Definitions	6-44
GetSet Custom Storage Class for Data Store Memory ..	6-46
Overview	6-46
Example of Generated Code with GetSet Custom Storage Class	6-48
Setting Code Generation Options for Custom Storage Classes	6-50
Custom Storage Class Limitations	6-51
Older Custom Storage Classes (Prior to Release 14) ...	6-52
Introduction	6-52
Simulink.CustomParameter Class	6-52
Simulink.CustomSignal Class	6-54
Instance-Specific Attributes for Older Storage Classes ...	6-57
Assigning a Custom Storage Class to Data	6-59
Code Generation with Older Custom Storage Classes	6-59
Compatibility Issues for Older Custom Storage Classes ..	6-60

7

Introduction to Memory Sections	7-2
Overview	7-2
Memory Sections Demo	7-2
Additional Information	7-2
Requirements for Defining Memory Sections	7-4
Defining Memory Sections	7-6
Editing Memory Section Properties	7-6
Specifying the Memory Section Name	7-7
Specifying a Qualifier for Custom Storage Class Data Definitions	7-8
Specifying Comment and Pragma Text	7-8
Surrounding Individual Definitions with Pragmas	7-9
Including Identifier Names in Pragmas	7-9
Applying Memory Sections	7-10
Assigning Memory Sections to Custom Storage Classes ..	7-10
Applying Memory Sections to Model-Level Functions and Internal Data	7-12
Applying Memory Sections to Atomic Subsystems	7-15
Examples of Generated Code with Memory Sections ..	7-18
Sample ERT-Based Model with Subsystem	7-18
Model-Level Data Structures	7-19
Model-Level Functions	7-21
Subsystem Function	7-22
Memory Section Limitation	7-23

Advanced Code Generation Techniques

8

Introduction	8-2
---------------------------	-----

Code Generation with User-Defined Data Types	8-4
Overview	8-4
Specifying Type Definition Location for User-Defined Data Types	8-5
Using User-Defined Data Types for Code Generation	8-6
Customizing the Target Build Process with the	
STF_make_rtw Hook File	8-7
Overview	8-7
File and Function Naming Conventions	8-7
STF_make_rtw_hook.m Function Prototype and Arguments	8-8
Applications for STF_make_rtw_hook.m	8-12
Using STF_make_rtw_hook.m for Your Build Procedure ..	8-13
Customizing the Target Build Process with	
sl_customization.m	8-14
Overview	8-14
Registering Build Process Hook Functions Using sl_customization.m	8-16
Variables Available for sl_customization.m Hook Functions	8-17
Example Build Process Customization Using sl_customization.m	8-17
Auto-Configuring Models for Code Generation	8-19
Overview	8-19
Utilities for Accessing Model Configuration Properties ...	8-19
Automatic Model Configuration Using ert_make_rtw_hook	8-20
Using the Auto-Configuration Utilities	8-22
Generating Efficient Code with Optimized ERT	
Targets	8-23
Overview	8-23
Default ERT Target	8-24
Optimized Fixed-Point ERT Target	8-24
Optimized Floating-Point ERT Target	8-26
Using the Optimized ERT Targets	8-28
Custom File Processing	8-31
Overview	8-31

Custom File Processing Components	8-32
Custom File Processing User Interface Options	8-32
Code Generation Template (CGT) Files	8-34
Using Custom File Processing (CFP) Templates	8-38
Custom File Processing (CFP) Template Structure	8-39
Generating Source and Header Files with a Custom File Processing (CFP) Template	8-40
Code Template API Summary	8-50
Generating Custom File Banners	8-53

Optimizing Your Model with Configuration Wizard

Blocks and Scripts	8-59
Overview	8-59
Configuration Wizards vs. Auto-Configuring Targets	8-61
Adding a Configuration Wizard Block to Your Model	8-61
Using Configuration Wizard Blocks	8-64
Creating a Custom Configuration Wizard Block	8-64

Replacement of STF_rtw_info_hook Mechanism

8-72

Optimizing Task Scheduling for Multirate Multitasking

Models on RTOS Targets	8-73
Overview	8-73
Using rtmStepTask	8-74
Task Scheduling Code for Multirate Multitasking Model on Wind River Systems VxWorks Target	8-74
Suppressing Redundant Scheduling Calls	8-75

Target Function Libraries

9

Introduction to Target Function Libraries	9-2
Overview of Target Function Libraries	9-2
Target Function Libraries General Workflow	9-5
Target Function Libraries Quick-Start Example	9-7
Creating Function Replacement Tables	9-14
Overview of Function Replacement Table Creation	9-14
Creating Table Entries	9-17

Example: Mapping Floating-Point Math Functions to Target-Specific Implementations	9-24
Example: Mapping the memcpy Function to a Target-Specific Implementation	9-29
Example: Mapping Nonfinite Support Utility Functions to Target-Specific Implementations	9-34
Example: Mapping Operators to Target-Specific Implementations	9-39
Mapping Fixed-Point Operators to Target-Specific Implementations	9-45
Specifying Build Information for Function Replacements	9-70
Adding Target Function Library Reserved Identifiers	9-74

Examining and Validating Function Replacement

Tables	9-76
Overview of Function Replacement Table Validation	9-76
Invoking the Table Definition M-File	9-76
Using the Target Function Library Viewer to Examine Your Table	9-77
Using the Target Function Library Viewer to Examine Registered TFLs	9-78
Tracing Code Generated Using Your Target Function Library	9-80
Examining TFL Cache Hits and Misses	9-82

Registering Target Function Libraries

Overview of TFL Registration	9-84
Using the sl_customization API to Register a TFL with Simulink Software	9-85
Using the rtwTargetInfo API to Register a TFL with Embedded MATLAB Coder Software	9-89
Registering Multiple TFLs	9-90

Target Function Library Limitations

ERT Target Requirements, Restrictions, and Control Files

10

Requirements and Restrictions for ERT-Based Simulink Models	10-2
ERT System Target File and Template Makefiles	10-4

Examples

A

Data Structures, Code Modules, and Program Execution	A-2
Code Generation	A-2
Verifying Generated Code	A-2
Custom Storage Classes	A-3
Memory Sections	A-3
Advanced Code Generation	A-3
Target Function Libraries	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 “Introduction to 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 the following information in the Real-Time Workshop documentation:

- *Real-Time Workshop Getting Started Guide*
- “Code Generation and the Build Process”
- “Building Subsystems and Working with Referenced Models”
- “Working with Data”
- “Optimizing a Model for Code Generation”
- *Real-Time Workshop Reference*

The following information might also be helpful:

- “Inserting Custom Code Into Generated Code”
- “Working with Embedded MATLAB™ Coder”
- “Asynchronous Support”
- “Models with Multiple Sample Rates”
- “Writing S-Functions for Real-Time Workshop Code Generation”
- “Data Exchange APIs”
- “Generating ASAP2 Files”
- “External Mode”
- “Timing Services”
- “Limitations on the Use of Absolute Time”
- “Glossary”
- “Troubleshooting”
- *Real-Time Workshop Target Language Compiler*

Data Structures, Code Modules, and Program Execution

This chapter describes the essential components and techniques used in a Real-Time Workshop Embedded Coder application. The topics include data structures, code modules, header files, generated program execution, and task management. For an introduction to the Real-Time Workshop Embedded Coder product and its capabilities, see *Real-Time Workshop Embedded Coder Getting Started Guide*. For information on applications of the Real-Time Workshop Embedded Coder product and how and when you might use it during system development, see “Introduction to Real-Time Workshop Technology” in the Real-Time Workshop documentation.

- “Real-Time Model (rtModel) Data Structure” on page 2-2
- “Code Modules” on page 2-4
- “Generating the Main Program Module” on page 2-8
- “Program Execution Overview” on page 2-10
- “Stand-Alone Program Execution” on page 2-11
- “Wind River Systems VxWorks Example Main Program Execution” on page 2-21
- “Model Entry Points” on page 2-24
- “Static Main Program Module” on page 2-26
- “Rate Grouping Compliance and Compatibility Issues” on page 2-31

Real-Time Model (rtModel) Data Structure

In this section...

“Overview” on page 2-2

“rtModel Accessor Macros” on page 2-3

Overview

The Real-Time Workshop Embedded Coder *real-time model* data structure, also referred to as `rtModel`, encapsulates information about the root model.

To reduce memory requirements, `rtModel` contains only information required by your model. For example, the fields related to data logging are generated only if the model has the **MAT-file logging** code generation option enabled. `rtModel` may also contain model-specific information related to timing, solvers, and model data such as inputs, outputs, states, and parameters.

By default, `rtModel` contains an error status field that your code can monitor or set. If you do not need to log or monitor error status in your application, select the **Suppress error status in real-time model data structure** option. This further reduces memory usage. Selecting this option may also cause `rtModel` to disappear completely from the generated code.

The symbol definitions for `rtModel` in generated code are as follows:

- Structure definition (in `model.h`):

```
struct RT_MODEL_model {  
    ...  
};
```

- Forward declaration typedef (in `model_types.h`):

```
typedef struct RT_MODEL_model RT_MODEL_model;
```

- Variable and pointer declarations (in `model.c` or `.cpp`):

```
RT_MODEL_model model_M_  
RT_MODEL_model *model_M = &model_M_;
```

- Variable export declaration (in *model.h*):

```
extern RT_MODEL_model *model_M;
```

rtModel Accessor Macros

To enable you to interface your code to `rtModel`, the Real-Time Workshop Embedded Coder software provides accessor macros. Your code can use the macros, and access the fields they reference, with *model.h*.

If you are interfacing your code to a single model, refer to its `rtModel` generically as *model_M*, and use the macros to access its `rtModel` as in the following code fragment.

```
#include "model.h"
const char *errStatus = rtmGetErrorStatus(model_M);
```

To interface your code to the `rtModel` structures of more than one model, simply include the *model.h* headers for each model, as in the following code fragment.

```
#include "modelA.h" /* Make model A entry points visible */
#include "modelB.h" /* Make model B entry points visible */

void myHandWrittenFunction(void)
{
    const char_T *errStatus;

    modelA_initialize(1); /* Call model A initializer */
    modelB_initialize(1); /* Call model B initializer */
    /* Refer to model A's rtModel */
    errStatus = rtmGetErrorStatus(modelA_M);
    /* Refer to model B's rtModel */
    errStatus = rtmGetErrorStatus(modelB_M);
}
```

To view macros related to `rtModel` that are applicable to your specific model, generate code with a code generation report (see “Creating and Using a Code Generation Report” on page 3-43). Then, view *model.h* by clicking the hyperlink in the report.

Code Modules

In this section...
“Introduction” on page 2-4
“Generated Code Modules” on page 2-4
“User-Written Code Modules” on page 2-7

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 them.

Note that in most cases, 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 let you view the generated code in the MATLAB® Help browser. See “Creating and Using a Code Generation Report” on page 3-43 for further information.

Generated Code Modules

The Real-Time Workshop Embedded Coder software creates a build directory in your working directory to store generated source code. The build directory also contains object files, a makefile, and other files created during the code generation process. The default name of the build directory is *model_ert_rtw*.

Real-Time Workshop® Embedded Coder™ File Packaging on page 2-5 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 non-embedded targets. See the Real-Time Workshop documentation for further information.

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 by the generated source files in the model. You do not need to include <i>model_private.h</i> when interfacing hand-written code to 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 by subsystem <i>.c</i> or <i>.cpp</i> files in the model. If you are interfacing your hand-written code to generated code for one or more models, you should include <i>model.h</i> for each model to which you want to interface.
<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 used for the model's 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. Note that these structures and zero representations are declared <i>extern</i> 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. These may be needed by function declarations of reusable functions. 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 require these definitions.
<i>ert_main.c</i> or <i>.cpp</i> (optional)	This file is generated only if the Generate an example main program option is on. (This option is on by default.) See “Generating the Main Program Module” on page 2-8.

Real-Time Workshop Embedded Coder File Packaging (Continued)

File	Description
<p>autobuild.h (optional)</p>	<p>This file is generated only if the Generate an example main program option is off. (See “Generating the Main Program Module” on page 2-8.)</p> <p>autobuild.h contains <code>#include</code> directives required by the static version of the <code>ert_main.c</code> main program module. Since the static <code>ert_main.c</code> is not created at code generation time, it includes <code>autobuild.h</code> to access model-specific data structures and entry points.</p> <p>See “Static Main Program Module” on page 2-26 for further information.</p>
<p><i>model_capi.c</i> or <i>.cpp</i> <i>model_capi.h</i> (optional)</p>	<p>Provides data structures that enable a running program to access model parameters and signals without use of external mode. To learn how to generate and use the <i>model_capi.c</i> or <i>.cpp</i> and <i>.h</i> files, see the “Data Exchange APIs” chapter in the Real-Time Workshop documentation.</p>

You can also customize the generated set of files in several ways:

- **Nonvirtual subsystem code generation:** You can 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. See “Nonvirtual Subsystem Code Generation” in the Real-Time Workshop documentation for further information.
- **Custom storage classes:** You can use custom storage classes to partition generated data structures into different files based on file names you specify. See Chapter 6, “Custom Storage Classes” for further information.
- **Module Packaging Features (MPF)** also lets you direct the generated code into a required set of *.c* or *.cpp* and *.h* files, and control the internal organization of the generated files. See the Module Packaging Features document for details.

User-Written Code Modules

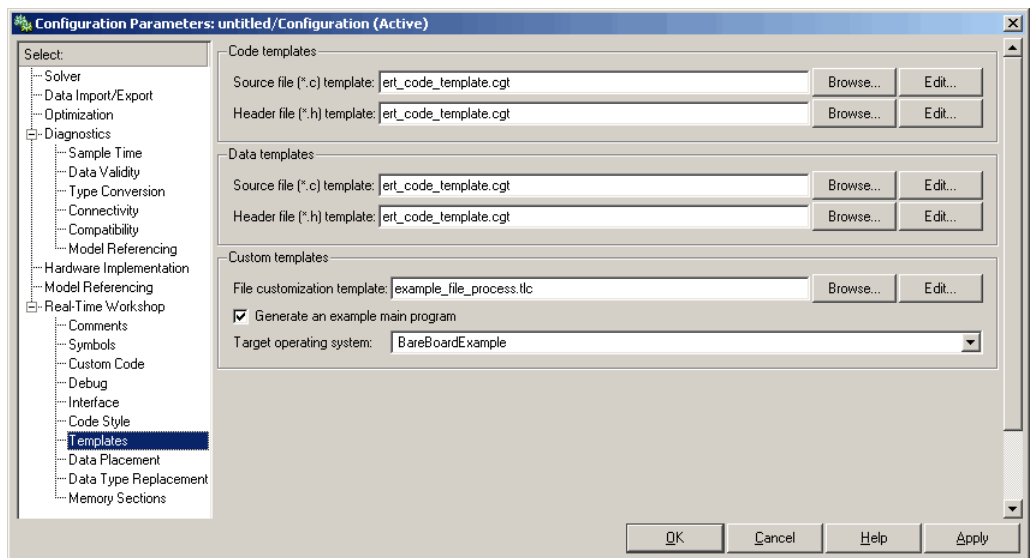
Code that you write to interface with generated model code usually includes a customized main module (based on a main program provided by the Real-Time Workshop software), and may also include interrupt handlers, device driver blocks and other S-functions, and other supervisory or supporting code.

You should establish a working directory for your own code modules. Your working directory should be 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. More extensive modifications to the ERT target files are needed if you want to generate code for a particular microprocessor or development board, and to deploy the code on target hardware with a cross-development system.

See the Developing Embedded Targets document for information on how to customize the ERT target for your production requirements.

Generating the Main Program Module

The **Generate an example main program** option controls whether or not `ert_main.c` or `ert_main.cpp` is generated for your Simulink® model. This option is located in the **Templates** pane of the Configuration Parameters dialog box, as shown in this figure.



Options for Generating a Main Program

By default, **Generate an example main program** is on. When **Generate an example main program** is selected, the **Target operating system** pop-up menu is enabled. This menu lets you choose the following options:

- **BareBoardExample**: Generate a bareboard main program designed to run under control of a real-time clock, without a real-time operating system.
- **VxWorksExample**: Generate a fully commented example showing how to deploy the code under the Wind River Systems VxWorks® real-time operating system.

Regardless of which **Target operating system** you select, `ert_main.c` or `.cpp` includes

- The `main()` function for the generated program
- Task scheduling code that determines how and when block computations execute on each time step of the model

The operation of the main program and the scheduling algorithm employed depend primarily upon whether your model is single-rate or multirate, and also upon your model's solver mode (`SingleTasking` vs. `MultiTasking`). These are described in detail in “Program Execution Overview” on page 2-10.

If you turn the **Generate an example main program** option off, the Real-Time Workshop Embedded Coder software provides a static version of the file `ert_main.c` as a basis for your custom modifications (see “Static Main Program Module” on page 2-26).

Note Once you have generated and customized the main program, you should take care to turn **Generate an example main program** off to prevent regenerating the main module and overwriting your customized version.

You can use a custom file processing (CFP) template file to override the normal main program generation, and generate a main program module customized for your target environment. To learn how to do this, see “Customizing Main Program Module Generation” on page 8-45.

Program Execution Overview

The following sections describe how programs generated by the Real-Time Workshop Embedded Coder software execute, from the top level down to timer interrupt level:

- “Stand-Alone Program Execution” on page 2-11 describes the operation of self-sufficient example programs that do not require an external real-time executive or operating system.
- “Wind River Systems VxWorks Example Main Program Execution” on page 2-21 describes the operation of example programs designed for deployment under the VxWorks real-time operating system.
- “Model Entry Points” on page 2-24 describes the model entry-point functions that are generated for both stand-alone and VxWorks example programs.

Stand-Alone Program Execution

In this section...
“Overview” on page 2-11
“Main Program” on page 2-12
“rt_OneStep” on page 2-14

Overview

By default, the Real-Time Workshop Embedded Coder software generates *stand-alone* programs that do not require an external real-time executive or operating system. A stand-alone program requires some minimal modification to be adapted to the target hardware; these modifications are described in the following sections. The stand-alone program architecture supports execution of models with either single or multiple sample rates.

To generate a stand-alone 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** pop-up menu is enabled. Select **BareBoardExample** from this menu (this option is the default selection).

The core of a stand-alone 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` function(s). 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.

The Real-Time Workshop Embedded Coder software generates significantly different code for multirate models depending on the following factors:

- Whether the model executes in singletasking 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.

Main Program

- “Overview of Operation” on page 2-12
- “Guidelines for Modifying the Main Program” on page 2-13

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.

- After calling `model_initialize`:
 - Initialize target-specific data structures and hardware such as ADCs or DACs.
 - Install `rt_OneStep` as a timer ISR.
 - Initialize timer hardware.
 - 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.

- Optionally, insert background task calls in the main loop.
- On termination of 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

- “Overview of Operation” on page 2-14
- “Single-Rate Singletasking Operation” on page 2-15
- “Multirate Multitasking Operation” on page 2-16
- “Multirate Singletasking Operation” on page 2-18
- “Guidelines for Modifying rt_OneStep” on page 2-19

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` vs. `MultiTasking`)

Permitted Solver Modes for Real-Time Workshop® Embedded Coder™ Targeted Models on page 2-14 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
<code>SingleTasking</code>	Allowed	Allowed
<code>MultiTasking</code>	Disallowed	Allowed
<code>Auto</code>	Allowed (defaults to <code>SingleTasking</code>)	Allowed (defaults to <code>MultiTasking</code>)

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 Singletasking 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 Sub-Rate 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 *sub-rate* 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 sub-rate 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 sub-rate 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 sub-rate 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 2-31 for information about how to adapt your code for rate grouping compatibility. This adaptation lets your multirate, multitasking models generate more efficient code.

Multirate Singletasking Operation

In a multirate singletasking program, by definition, all sample times in the model must be an integer multiple of the model’s fixed-step size.

In a multirate singletasking 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 singletasking 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 sub-rate 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 re-enable interrupts after the overrun flag(s) 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 sub-rates before calling `model_stepN` in the sub-rate loop.
- In a multirate, multitasking model, get model outputs associated with sub-rates after calling `model_stepN` in the sub-rate 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.

Please 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.

Wind River Systems VxWorks Example Main Program Execution

In this section...

“Introduction to the VxWorks Example Main Program” on page 2-21

“Task Management” on page 2-22

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 “Stand-Alone Program Execution” on page 2-11, 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** pop-up 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

- “Overview of Operation” on page 2-22
- “Single-Rate Singletasking Operation” on page 2-22
- “Multirate Multitasking Operation” on page 2-23
- “Multirate Singletasking Operation” on page 2-23

Overview of Operation

In a VxWorks example program, the main program and the base rate and sub-rate 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 sub-rate 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 2-14). The following sections discuss each possible case.

Single-Rate Singletasking Operation

In a single-rate, singletasking 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 sub-rate 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 sub-rate 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 Singletasking Operation

In a multirate, singletasking 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.

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 “Configuring Model Interfaces” on page 3-21).

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 2-26
“Rate Grouping and the Static Main Program” on page 2-27
“Modifying the Static Main Program” on page 2-28

Overview

In most cases, the easiest strategy for deploying your generated code is to use the **Generate an example main program option** to generate the `ert_main.c` or `.cpp` module (see “Generating the Main Program Module” on page 2-8).

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 “Stand-Alone Program Execution” on page 2-11. 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 2-13 and “Guidelines for Modifying `rt_OneStep`” on page 2-19.

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.
- Remove old `#include ertformat.h` directives. `ertformat.h` will be obsoleted in a future release. The following macros, formerly defined in `ertformat.h`, are now defined within `ert_main.c`:

```
EXPAND_CONCAT
CONCAT
MODEL_INITIALIZE
MODEL_STEP
MODEL_TERMINATE
MODEL_SETEVENTS
RT_OBJ
```

See also the comments in `ertformat.h`.

- 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 2-19.

- 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 2-31

“Making Your S-Functions Rate Grouping Compliant” on page 2-31

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 2-27 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. Non-compliant 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 non-compliant 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 ($0 \dots \text{NumTasks} - 1$).
- 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
```


Code Generation Options and Optimizations

- “Accessing the ERT Target Options” on page 3-2
- “Support for Continuous Time Blocks, Continuous Solvers, and Stop Time” on page 3-4
- “Mapping Application Requirements to Configuration Options” on page 3-6
- “Configuring a Model” on page 3-7
- “Tips for Optimizing the Generated Code” on page 3-33
- “Creating and Using a Code Generation Report” on page 3-43
- “Automatic S-Function Wrapper Generation” on page 3-68
- “Exporting Function-Call Subsystems” on page 3-72
- “Nonvirtual Subsystem Modular Function Code Generation” on page 3-83
- “Controlling Model Function Prototypes” on page 3-96
- “Generating and Controlling C++ Encapsulation Interfaces” on page 3-118
- “Creating and Using Host-Based Shared Libraries” on page 3-140

Accessing the ERT Target Options

In this section...
“Introduction” on page 3-2
“Viewing ERT Target Options in the Configuration Parameters Dialog Box or Model Explorer” on page 3-3

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 Target Support Package™ FM5 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 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).

Support for Continuous Time Blocks, Continuous Solvers, and Stop Time

In this section...
“Generating Code for Continuous Time Blocks” on page 3-4
“Generating Code that Supports Continuous Solvers” on page 3-4
“Generating Code that Honors a Stop Time” on page 3-5

Generating Code for Continuous Time Blocks

The ERT target supports code generation for 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 at the MATLAB command line:

```
showblockdatatypetable
```

Then, refer to the “Code Generation Support” column of the table.

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 the Developing Embedded Targets document.

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 `ert_main.c`. `ert_main.c` controls the overall model code execution by calling the model step and initialization functions, 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.

Mapping Application Requirements to Configuration Options

The first step to applying Real-Time Workshop Embedded Coder configuration options to the application development process is to consider how your application requirements, particularly with respect to traceability, efficiency, 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 priority for your application — debugging, traceability, efficiency, extra safety precaution, or some other criteria?
- What is the second highest priority?
- Can the priority at the start of the project differ from the priority required for the end result? What tradeoffs can be made?

Once you have answered these questions, review “Mapping of Application Requirements to the Optimization Pane”. This table maps requirements of debugging, traceability, efficiency, and safety precautions to configuration parameters that are available for the Embedded Real-Time (ERT) target.

Configuring a Model

In this section...

- “Selecting an ERT Target” on page 3-7
- “Generating a Report that Includes Hyperlinks for Tracing Code to Model Blocks” on page 3-9
- “Customizing Comments in Generated Code” on page 3-10
- “Customizing Generated Identifiers” on page 3-12
- “Configuring Symbols” on page 3-13
- “Configuring Model Interfaces” on page 3-21
- “Controlling Code Style” on page 3-26
- “Configuring Templates for Customizing Code” on page 3-27
- “Configuring the Placement of Data in Code” on page 3-28
- “Configuring Replacement Data Types” on page 3-28
- “Configuring Memory Sections” on page 3-30
- “Configuring Optimizations” on page 3-31

Selecting an ERT Target

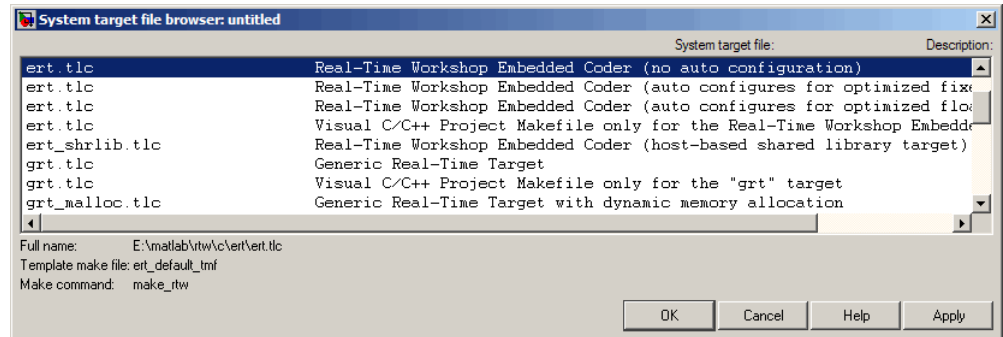
The **Browse** button in the **Target Selection** subpane of the **Real-Time Workshop > General** pane lets you select an ERT target with the System Target File Browser. See “Choosing and Configuring Your Target” in the Real-Time Workshop documentation for a general discussion of target selection.

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.



The optimized ERT target variants are discussed in detail in “Generating Efficient Code with Optimized ERT Targets” on page 8-23.

You can implement a custom auto-configuring target, using the same mechanism used by the optimized ERT target variants. “Auto-Configuring Models for Code Generation” on page 8-19 discusses the auto-configuration mechanism and utilities used by the optimized ERT target variants.

You can use the `ert_shrplib.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 Microsoft® Windows® dynamic link library (.dll) file or a UNIX^{®1} 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 3-140.

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

Generating a Report that Includes Hyperlinks for Tracing Code to Model Blocks

Real-Time Workshop Embedded Coder software extends the HTML code generation report that gets generated when you select the **Create code generation report** parameter. When you select this parameter for an ERT target, the parameters **Code-to-model** and **Model-to-code** appear. If you select these additional parameters, the HTML report includes hyperlinks from the code to the generating blocks in the model and right-clicking on the blocks in the model brings you to the code for that block. You can use these links to verify traceability of the generated code to the model.

For very large models (containing over 1000 blocks) generation of the hyperlinks can be time consuming. Therefore, if you do not have a need for traceability or after verifying the traceability of you generated code, consider disabling the parameter to speed up code generation.

For more information, see “Code-to-model” and “Model-to-code” in the Real-Time Workshop reference documentation.

Customizing Comments in 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 blockname 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 an M-code or TLC function.	Custom comments (MPT objects only).
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 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.)

- 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.

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, 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 3-13 and for limitations, see “Identifier Format Control Parameters Limitations” on page 3-19).
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 3-16).
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 #define 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 3-13
- “Specifying Identifier Formats” on page 3-13
- “Name Mangling” on page 3-16
- “Traceability” on page 3-17
- “Minimizing Name Mangling” on page 3-18
- “Model Referencing Considerations” on page 3-18
- “Exceptions to Identifier Formatting Conventions” on page 3-19
- “Identifier Format Control Parameters Limitations” on page 3-19

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 <code>Define</code>	#define naming parameter

For more information on these parameters, see “Specifying Simulink Data Object Naming Rules” 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 3-14. You can use or omit tokens as you want, with the exception of the \$M token, which is required (see “Name Mangling” on page 3-16) and subject to the use and ordering restrictions noted in Identifier Format Control Parameter Values on page 3-15.
- 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 3-14) 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 3-16). 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 3-18). 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>

Identifier Format Control Parameter Values on page 3-15 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, and \$A are disallowed.
Global types	\$N\$R\$M	\$N, \$R, \$M	\$F, \$H, and \$A are disallowed.
Field name of global types	\$N\$M	\$N, \$M, \$H, \$A	\$R and \$F are disallowed.
Subsystem methods	\$R\$N\$M\$F	\$R, \$N, \$M, \$F, \$H	\$F and \$H are empty for Stateflow functions; \$A is disallowed.

Identifier Format Control Parameter Values (Continued)

Parameter	Default Value	Supported Tokens	Restrictions
Local temporary variables	\$N\$M	\$N, \$M, \$R	\$F, \$H, and \$A are disallowed.
Local block output variables	rtb_{\$N\$M}	\$N, \$M, \$A	\$R, \$F, and \$H are disallowed.
Constant macros	\$\$N\$M	\$R, \$N, \$M	\$F, \$H, and \$A are disallowed.

Non-ERT based targets (such as the GRT target) implicitly use a default \$\$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 3-16).

For limitations that apply to **Identifier format control** parameters, see “Identifier Format Control Parameters Limitations” on page 3-19.

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 \$\$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 3-17 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
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 3-14. 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 auto-generated 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 **\$R\$F**, 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 the Signal Processing Blockset product, that rely on S-functions
- Local variables generated by S-functions or by add-on products, such as the Signal Processing Blockset product, 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 product, that rely on S-functions
 - Local variables generated by S-functions or by add-on products, such as Signal Processing Blockset product, 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 auto-generated identifiers.

- Identifiers defined in custom code
- Identifiers defined in custom header files
- Identifiers introduced through a non-ANSI² 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.

Configuring Model Interfaces

In addition to the interface parameters available for GRT targets, you can configure the following for ERT targets:

- “Configuring Support for Numeric Data” on page 3-21
- “Configuring Support for Time Values” on page 3-22
- “Configuring Support for Non-Inlined S-Functions” on page 3-22
- “Configuring Model Function Generation and Argument Passing” on page 3-23
- “Configuring a Model for Code Reuse” on page 3-24

Configuring Support for Numeric Data

By default, ERT targets support code generation for integer, floating-point, non-finite, 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 .

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

To Generate Code that Supports...	Do...
Non-finite 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 “Timing Services” 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.

Configuring Support for Non-Inlined S-Functions

To generate code for non-inlined S-functions in a model, select **Support non-inlined S-functions**. The generation of non-inlined S-functions requires floating-point and non-finite 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 “Configuring a Model for Code Reuse” on page 3-24 for details.

To...	Do...
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 2-24 for information on the calling interface generated for model functions in this case.
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>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>
Launch the Model Interface dialog box (see “Configuring Model Function Prototypes” on page 3-97) preview and modify the model’s step and initialization function prototypes	Click Configure Model Functions . Based on the Function specification value you select for your model initialization and step functions, 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 “Controlling Model Function Prototypes” on page 3-96.

For more information, see “Real-Time Workshop Pane: Interface” in the Real-Time Workshop reference documentation.

Configuring a Model 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.

Controlling Code Style

You can control the following style aspects in generated code:

- Level of parenthesization
- Whether operand order is preserved in expressions
- Whether conditions are preserved in `if` statements

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

Code and data templates provide a way to customize generated code.

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 can be the same template file that you specify for Source file (.c) template , in which case identical banners are generated in source and header files. The default template is <i>matlabroot</i> /toolbox/rtw/targets/ecoder/ert_code_template.cgt.
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 as required. See “Custom File Processing” on page 8-31 for detailed information.
Generate a model-specific example main program module	Select Generate an example main program . See “Generating the Main Program Module” on page 2-8 for more information.

Template files that you specify must be located on the MATLAB path.

For more detail, see the Module Packaging Features document. See also “Generating Custom File Banners” on page 8-53 for a simple example of how a code template can be applied to generate customized comment sections in generated code files.

Configuring the Placement of Data in 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.

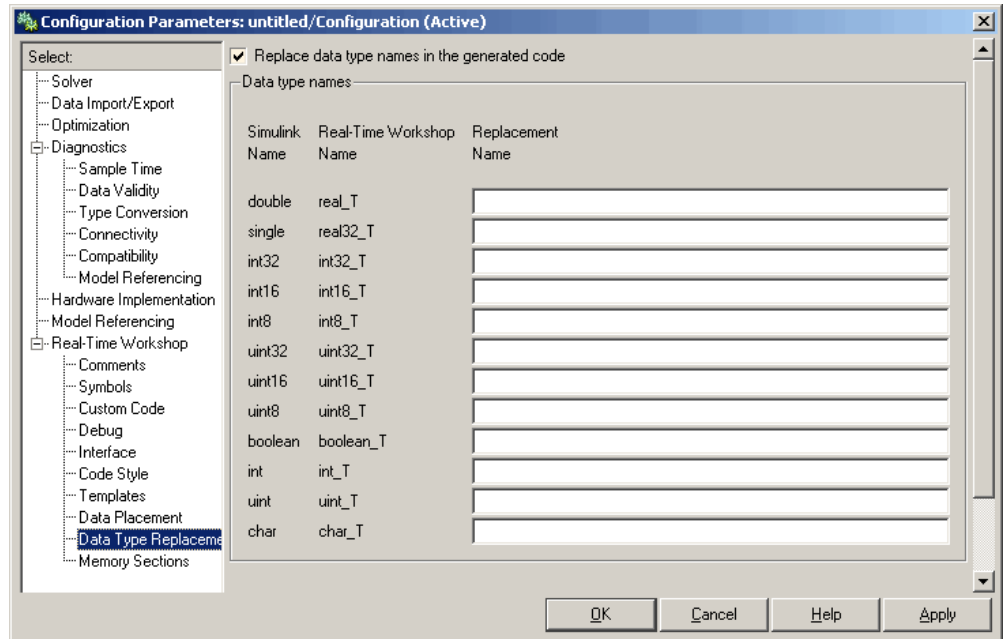
For details data placement, see the Module Packaging Features document.

Configuring Replacement Data 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`, `uint8`, and `boolean`, the replacement data type's `BaseType` must match the built-in data type.
- For `int`, `uint`, and `char`, the replacement data type's size must match the size displayed for `int` or `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” in the Module Packaging Features document.

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**.

For details on using memory sections, see Chapter 7, “Memory Sections”.

Configuring 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 .
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 data zero initialization .
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.

To...	Select or Specify...
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.
Remove code that ensures that execution of the generated code produces the same results as simulation when mapping from NaN to integer zero occurs	<p>Select Remove code from floating-point to integer conversions with saturation that maps NaN to zero. 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 NaN values.</p> <hr/> <p>Note This parameter is enabled when you select the floating-point numbers and non-finite numbers check boxes in the Real-Time Workshop > Interface pane.</p> <hr/>
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 “Timing Services”, “Using Timers in Asynchronous Tasks”, and “Controlling Memory Allocation for Time Counters” in the Real-Time Workshop documentation.

Tips for Optimizing the Generated Code

In this section...

- “Introduction” on page 3-33
- “Using Auto-Optimized Targets” on page 3-33
- “Using Configuration Wizard Blocks” on page 3-34
- “Setting Hardware Implementation Parameters Correctly” on page 3-34
- “Removing Unnecessary Initialization Code” on page 3-36
- “Generating Pure Integer Code If Possible” on page 3-37
- “Disabling MAT-File Logging” on page 3-37
- “Using Virtualized Output Ports Optimization” on page 3-38
- “Using Stack Space Allocation Options” on page 3-39
- “Using External Mode with the ERT Target” on page 3-41

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 a Model for Code Generation” section of the Real-Time Workshop documentation for optimization techniques that are common to all target configurations.

Using Auto-Optimized Targets

To make it easier for you to generate the most efficient code for your target CPU, the Real-Time Workshop Embedded Coder software provides two auto-optimized ERT target variants. These target variants are optimized, respectively, for fixed-point and floating-point code generation.

Before generating and deploying code, consider using one of these optimized target variants. The optimized ERT target variants are discussed in detail in “Generating Efficient Code with Optimized ERT Targets” on page 8-23.

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 M-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 8-59 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 “Describing Hardware Properties” 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 data 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 outports 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 data zero initialization**, it is not guaranteed that memory is in a known state each time the generated code begins execution. If you select the parameter, 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 data 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 data 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 data 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

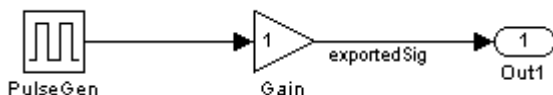
Clear the **MAT-file logging** option in the **Verification** section of the **Interface** pane. This setting is the default, and is recommended for embedded applications because it eliminates the extra code and memory usage required to initialize, update, and clean up logging variables. In addition 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 3-38 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;

/* Outport 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 “Working with Data” section of the Real-Time Workshop documentation.

Using Stack Space Allocation Options

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. This section discusses options that

- Let you control whether signal storage is declared in global memory space, or locally in functions (that is, in stack variables).
- Control the allocation of stack space when using local storage.

For a complete discussion of signal storage options, see the “Working with Data” 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** tab of the Configuration Parameters dialog box. Make sure that the **Signal storage reuse** option is selected. If **Signal storage reuse** is off, the **Enable local block outputs** option is not available.
- 2** Select the **Enable local block outputs** option. Click **Apply** if necessary.

Your embedded application may be constrained by limited stack space. When the **Enable local block outputs** option is on, you can limit the use of stack space by using the following TLC variables:

- **MaxStackSize**: The total allocation size of local variables that are declared by all block outputs in this model cannot exceed **MaxStackSize** (in bytes). **MaxStackSize** can be any positive integer. If the total size of local block output variables exceeds this maximum, the remaining block output variables are allocated in global, rather than local, memory. The default value for **MaxStackSize** is `rtInf`, that is, unlimited stack size.

Note Local variables in the generated code from sources other than local block outputs and stack usage from sources such as function calls and context switching are not included in the **MaxStackSize** calculation. For overall executable stack usage metrics, you should do a target-specific measurement, such as using runtime (empirical) analysis or static (code path) analysis with object code.

- **MaxStackVariableSize**: Limits the size of any local block output variable declared in the code to `N` bytes, where `N > 0`. A variable whose size exceeds **MaxStackVariableSize** is allocated in global, rather than local, memory. The default is 4096.

To set either of these variables, use `assign` statements in the system target file (`ert.tlc`), as in the following example.

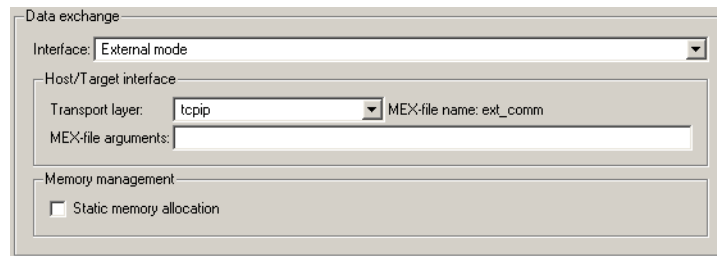
```
%assign MaxStackSize = 4096
```

You should write your `%assign` statements in the `Configure RTW code generation settings` section of the system target file. The `%assign` statement is described in the `Target Language Compiler` document.

Using External Mode with the ERT Target

Selecting the **External mode** option turns on generation of code to support external mode communication between host (the Simulink model) and target systems. Real-Time Workshop Embedded Coder software supports all features of Simulink external mode, as described in the “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

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

$$\text{stop time in seconds} / \text{base rate step size in seconds}$$

Creating and Using a Code Generation Report

In this section...

“Overview” on page 3-43

“Generating an HTML Code Generation Report” on page 3-44

“Using Code-to-Model Traceability” on page 3-46

“Using Model-to-Code Traceability” on page 3-48

“Using the Model-to-Code Navigation Dialog Box to Load Existing Trace Information” on page 3-50

“Using the Code Interface Report to Analyze the Generated Code Interface” on page 3-51

“Using the Traceability Report to Completely Map Model Elements to Code” on page 3-63

“HTML Code Generation Report Limitations” on page 3-65

Overview

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 and date information. The **Configuration Settings at the Time of Code Generation** link opens a noneditable view of the Configuration Parameters dialog 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 Source 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,

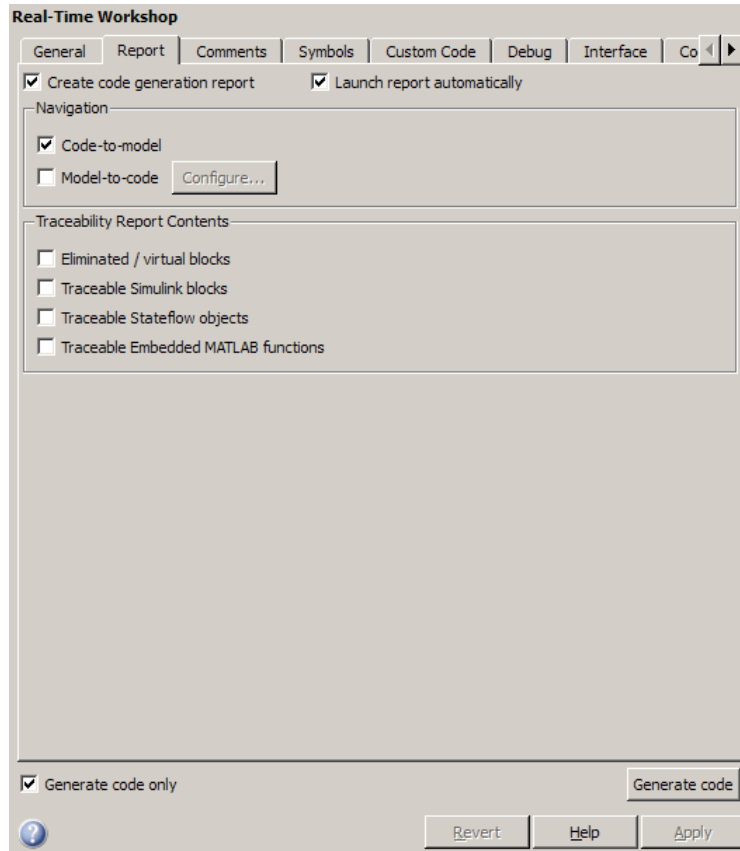
- Global variable instances are hyperlinked to their definitions.
- If you selected the traceability option **Code-to-model**, hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click on the hyperlinks to view the relevant blocks or subsystems in a Simulink model window.
- If you selected the traceability option **Model-to-code**, you can view the generated code for any block in the model. To highlight a block's generated code in the HTML report, right-click the block and select **Real-Time Workshop > Navigate to 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 as desired.



- 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 “Using Code-to-Model Traceability” on page 3-46.
- 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 “Using Model-to-Code Traceability” on page 3-48 and “Using the Traceability Report to Completely Map Model Elements to Code” on page 3-63.

Notes

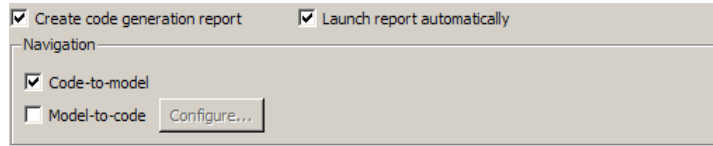
- For large models (containing over 1000 blocks), you may find that HTML report generation takes longer than you want. In this case, consider clearing the **Code-to-model** and **Model-to-code** check boxes. The report will be generated faster.
- You can also view the HTML report files, as well as the generated code files, in the Simulink Model Explorer. See “Viewing Generated Code in Model Explorer” in the Real-Time Workshop documentation for details.

Using Code-to-Model Traceability

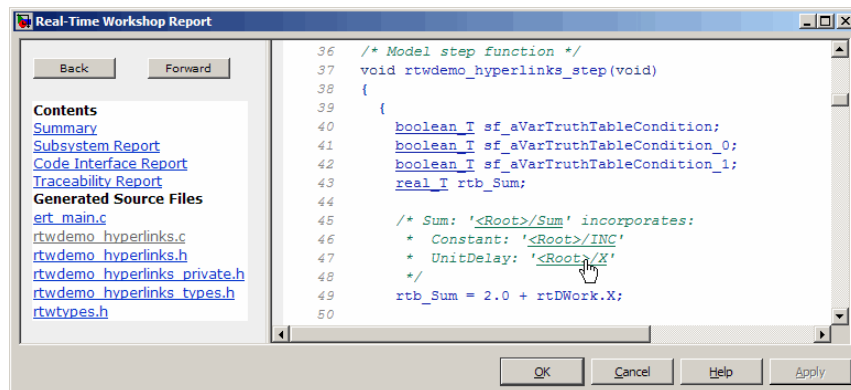
To use **Code-to-model**,

- 1 Open an ERT-based model and go to the **Real-Time Workshop→Report** pane of the Configuration Parameters dialog box. Select the option **Create**

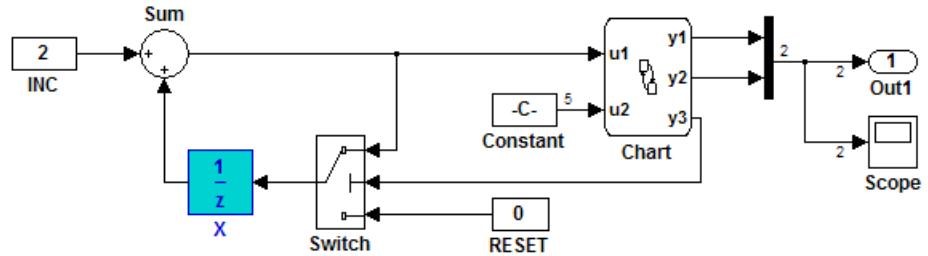
code generation report if it is not already selected. By default, **Launch report automatically** and **Code-to-model** also are selected.



- 2 Build or generate code for your model. This will launch an HTML code generation report.
- 3 In the HTML report window, click any of the hyperlinks present to highlight the source block. For example, in the HTML report shown below for the demo model `rtwdemo_hyperlinks`, you could click the first `UnitDelay` hyperlink in the generated code for the model step function.



Clicking the `UnitDelay` hyperlink highlights the corresponding source block in the Simulink model window.



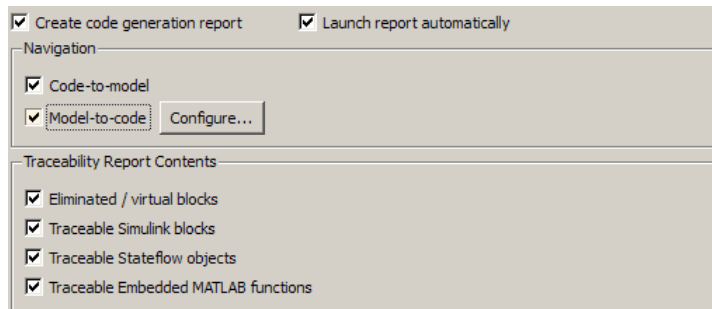
See also the demo `rtwdemo_hyperlinks`, which walks you through using **Code-to-model**.

Using Model-to-Code Traceability

To use **Model-to-code**,

- 1 Open an ERT-based model and go to the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box. Select the options **Create code generation report**, **Launch report automatically**, and **Model-to-code**, if they are not already selected.

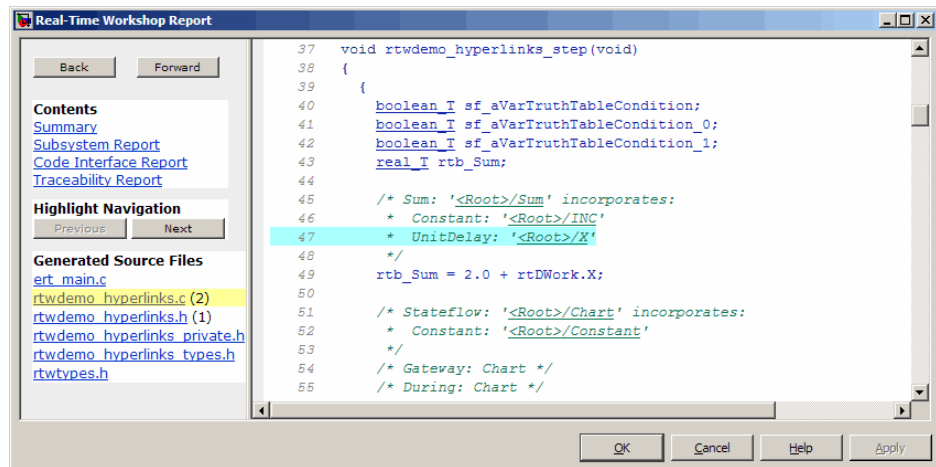
Selecting **Model-to-code** enables the **Configure** button, which you can use to open the Model-to-code navigation dialog box; see “Using the Model-to-Code Navigation Dialog Box to Load Existing Trace Information” on page 3-50. Selecting **Model-to-code** also enables and selects content options for generating a **Traceability Report**, described in “Using the Traceability Report to Completely Map Model Elements to Code” on page 3-63.



- 2 Build or generate code for your model. This launches an HTML code generation report.
- 3 In the model window, right-click any block. In the right-click menu, select **Real-Time Workshop > Navigate to Code**.

Selecting **Navigate to Code** highlights the generated code for the block in the HTML code generation report and takes you to the first instance. The total number of highlighted lines is displayed next to each source file name in the left panel of the HTML report. To navigate through multiple instances of highlighted lines, use the **Previous** and **Next** buttons.

The report below shows the result of tracing the Unit Delay block in the demo model `rtwdemo_hyperlinks`.



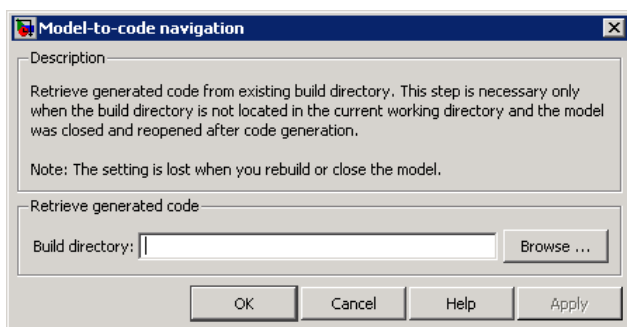
See also the demo `rtwdemo_hyperlinks`, which walks you through using **Model-to-code**.

Note If you later close and reopen your model, you may find that you cannot use right-click/**Real-Time Workshop** > **Navigate to Code** to trace a block's code, because **Navigate to Code** is greyed out. This means that a build directory for your model cannot be found in the current working directory. To address this you can do any 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 the **Configure** button and reload the model's trace information using the Model-to-code navigation dialog box. See “Using the Model-to-Code Navigation Dialog Box to Load Existing Trace Information” on page 3-50.

Using the Model-to-Code Navigation Dialog Box to Load Existing Trace Information

To open the Model-to-code navigation dialog box, click the **Configure** button on the **Real-Time Workshop** > **Report** pane of the Configuration Parameters dialog box. You can use this dialog box to reconnect your model with a previously-generated build directory, including trace information for model-to-code highlighting.



If you find that you cannot use right-click **Real-Time Workshop** > **Navigate to Code** to trace a block's code, because **Navigate to Code** is greyed out, it

means that a build directory for your model cannot be found in the current working directory. To fix this without having to reset the current working directory or rebuild the model,

- 1** Click the **Configure** button to launch the Model-to-code navigation dialog box.
- 2** In the dialog box, click the **Browse** button, browse to the build directory for your model, and select the directory. The build directory path should be displayed in the **Build directory** field of the dialog box, as shown in the example above.
- 3** Click **Apply** or **OK**. This loads trace information from the earlier build into your Simulink session, provided that you selected **Model-to-code** for the build.
- 4** Now you can now successfully use right-click **Real-Time Workshop > Navigate to Code** to trace a block's code.

Using the Code Interface Report to Analyze the Generated Code Interface

- “Code Interface Report Overview” on page 3-51
- “Generating a Code Interface Report” on page 3-52
- “Navigating Code Interface Report Subsections” on page 3-55
- “Interpreting the Entry Point Functions Subsection” on page 3-55
- “Interpreting the Inports and Outports Subsections” on page 3-59
- “Interpreting the Interface Parameters Subsection” on page 3-61
- “Interpreting the Data Stores Subsection” on page 3-62

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 3-66

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).



For help navigating the content of the code interface report subsections, see “Navigating Code Interface Report Subsections” on page 3-55. For help interpreting the content of the code interface report subsections, see the sections beginning with “Interpreting the Entry Point Functions Subsection” on page 3-55.

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 on 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.
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_fcnpctctrl</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_fcnpctctrl`.

Function: [rtwdemo_fcnprotoctrl_step_custom](#)

Prototype	boolean_T rtwdemo_fcnprotoctrl_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_fcnprotoctrl.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.

Field	Description
Code Identifier	<p>Lists the identifier associated with the inport or outport 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 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 <code><Root>/DS1</code>. 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 <code>Simulink.Signal</code> object, the field displays the name of the <code>Simulink.Signal</code> object prefixed with the label 'Global:'.

Field	Description
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. • 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]).

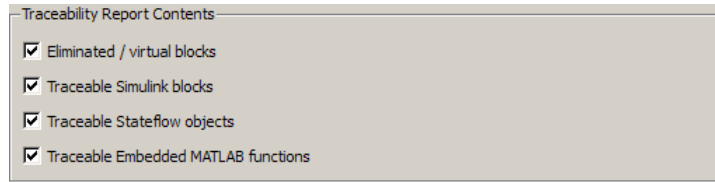
For example, here is the **Data Stores** subsection for the demo model `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the demo model window).

Data Stores

Data Store Source	Code Identifier	Data Type	Dimension
<Root>/Data Store Memory	mode	boolean_T	1

Using the Traceability Report to Completely Map Model Elements to Code

When you select the **Model-to-code** traceability option discussed in “Using Model-to-Code Traceability” on page 3-48, the generated HTML report includes a **Traceability Report** section. The content of the generated traceability report is determined by your selections among the following options on the **Real-Time Workshop > Report** pane of the Configuration Parameters dialog box.



The traceability report allows you to account for **Eliminated / virtual blocks** that are untraceable, versus the listed **Traceable Simulink blocks**, **Traceable Stateflow objects**, and **Traceable Embedded MATLAB functions**, providing a complete mapping between model elements and code.

The display below shows the beginning of the traceability report generated by selecting all four traceability content options for the demo 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

For the full report, see the demo [rtwdemo_hyperlinks](#), which walks you through using the traceability report, and be sure to select all four traceability content options.

HTML Code Generation Report Limitations

The following limitations apply to Real-Time Workshop Embedded Coder HTML code generation reports.

Traceability Limitations

- If a block name in your model contains a single quote ('), code-to-model and model-to-code are disabled for that block.

- If an asterisk (*) in a block name in your model causes a name-mangling ambiguity relative to other names in the model, code-to-model highlighting and model-to-code highlighting are disabled for that block. This is most likely to occur if an asterisk precedes or follows a slash (/) in a block name or appears at the end of a block name.
- If a block name in your model contains the character `(char(255))`, code-to-model highlighting and model-to-code highlighting are disabled for that block.
- Some types of subsystems are not traceable using **Model-to-code** at the subsystem block level:
 - Virtual subsystems
 - Masked subsystems
 - Nonvirtual subsystems for which code has been optimized away

If you cannot trace a subsystem at subsystem level, you may be able to trace individual blocks within the subsystem.

Code Interface Report Limitations

- 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.

Automatic S-Function Wrapper Generation

In this section...

“Overview” on page 3-68

“Generating an ERT S-Function Wrapper” on page 3-69

“S-Function Wrapper Generation Limitations” on page 3-71

Overview

An S-function wrapper is an S-function that calls your C or C++ code from within a Simulink model. S-function wrappers provide a standard interface between a Simulink model 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 using Simulink software), as well as for simulation acceleration purposes. For a complete description of wrapper S-functions, see the Simulink Writing S-Functions document.

Using the Real-Time Workshop Embedded Coder **Create Simulink (S-Function) block** option, you can build, in one automated step:

- A non-inlined C or C++ MEX S-function wrapper that calls Real-Time Workshop Embedded Coder generated code
- A model containing the generated S-function block, ready for use with other blocks or models

When the **Create Simulink (S-Function) block** option is on, 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 in a Simulink model.

The build process then compiles and links *model_sf.c* or *.cpp* with *model.c* or *.cpp* and the other 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

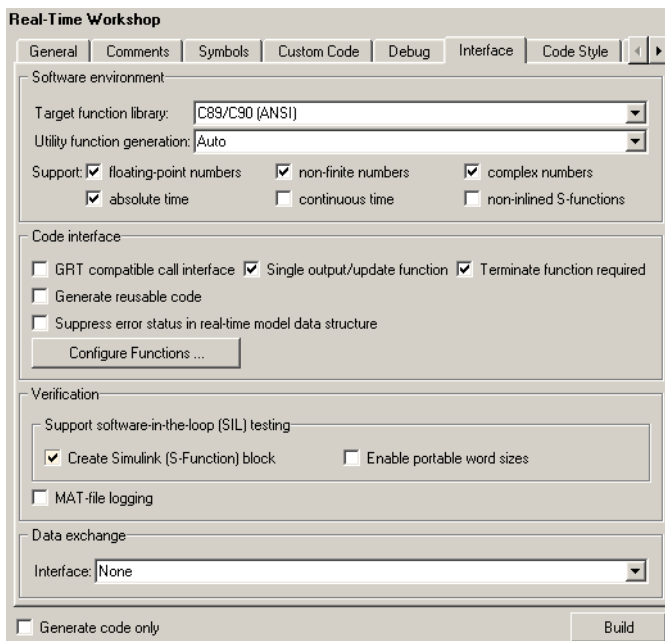
Real-Time Workshop build process creates and opens an untitled model containing the generated S-Function block.

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 **Use Embedded Coder** and click **Build**.

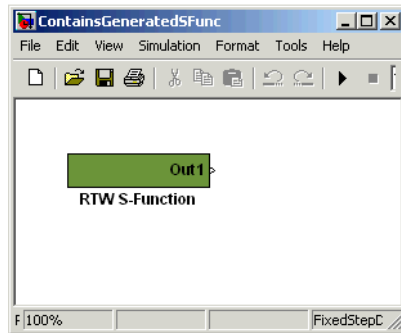
Generating an ERT S-Function Wrapper

To generate an 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 **Interface** pane.
- 3 Select the **Create Simulink (S-Function) block** option, as shown in this figure.



- 4 Configure the other code generation options as required.
- 5 To ensure that memory for the S-Function is initialized to zero, you should deselect the following options in the **Data Initialization** subpane of the **Optimization** 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 the **Build** button.
- 7 When the build process completes, an untitled model window opens. This model contains the generated S-Function block.



- 8 Save the new model.
- 9 The generated S-Function block is now ready to use with other Simulink blocks or models.

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 generating a Real-Time Workshop Embedded Coder S-function wrapper.
- Models that contain S-function blocks for which the S-function is not inlined with a TLC file are not supported when generating a Real-Time Workshop Embedded Coder S-function wrapper.
- You cannot use multiple instances of a generated Real-Time Workshop Embedded Coder S-function 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 S-function wrappers can be used with other blocks and models for such purposes as SIL code verification and simulation acceleration, but cannot be used for code generation.
- A MEX S-function wrapper must only be used in the MATLAB version in which the wrapper is created.

Exporting Function-Call Subsystems

In this section...

“Overview” on page 3-72

“Exported Subsystems Demo” on page 3-73

“Additional Information” on page 3-73

“Requirements for Exporting Function-Call Subsystems” on page 3-73

“Techniques for Exporting Function-Call Subsystems” on page 3-75

“Optimizing Exported Function-Call Subsystems” on page 3-76

“Exporting Function-Call Subsystems That Depend on Elapsed Time” on page 3-76

“Function-Call Subsystem Export Example” on page 3-77

“Function-Call Subsystems Export Limitations” on page 3-81

Overview

The 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
- Optionally generate an ERT S-function wrapper for 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 3-73. For limitations that apply, see “Function-Call Subsystems Export Limitations” on page 3-81.

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”
- *Writing 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 3-76 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 a scalar, a vector, 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 3-73.
- 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 an ERT S-function wrapper for the generated code, go to the **Interface** pane and select **Create Simulink (S-function) 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. The Real-Time Workshop build process generates code and places it in the working directory.

If you checked **Create Simulink (S-function) block** in step 2b, The build process 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

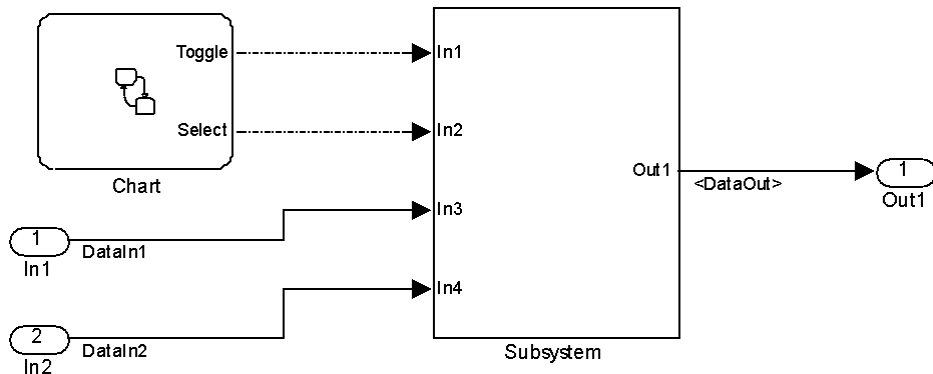
Elapsed Time Computation” in the Real-Time Workshop documentation for more information.

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,

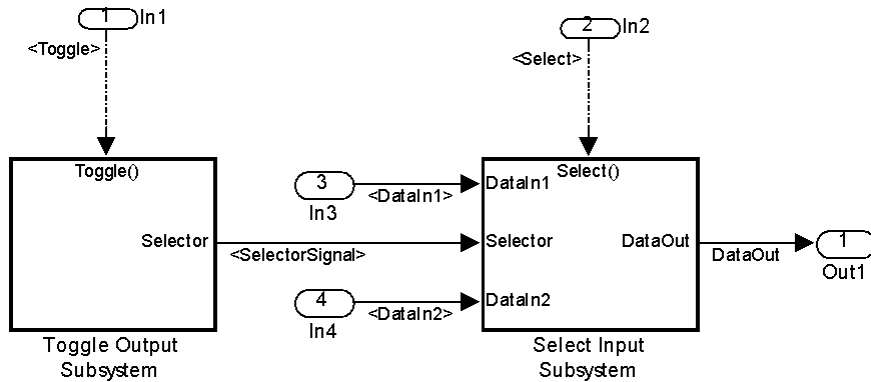
- 1 Right-click the trigger port block in the function-call subsystem and choose **TriggerPort Parameters** from the context menu.
- 2 Specify **periodic** in the **Sample time type** field.
- 3 Set the **Sample time** to the same granularity specified (directly or by inheritance) in the function-call 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.
- This release 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.
- This release supports code generation for ERT generated S-function blocks if the block does not have function-call input ports, but the ERT S-function block will appear as a noninlined S-function in the generated code.
- This release supports an ERT generated S-function block in accelerator mode only if its function-call initiator is noninlined in accelerator mode. Examples of noninlined initiators include all Stateflow charts.
- The ERT S-function wrapper 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 this release does not support the ERT S-function wrapper for an asynchronous system.
- This release does not support code generation for an ERT generated S-function block if the block was generated as a wrapper for exported function calls.
- The output port of an ERT generated S-function block cannot be merged using the Merge block.
- This release 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

In this section...

“Overview” on page 3-83

“Configuring Nonvirtual Subsystems for Generating Modular Function Code” on page 3-84

“Examples of Modular Function Code for Nonvirtual Subsystems” on page 3-88

“Nonvirtual Subsystem Modular Function Code Limitations” on page 3-94

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.

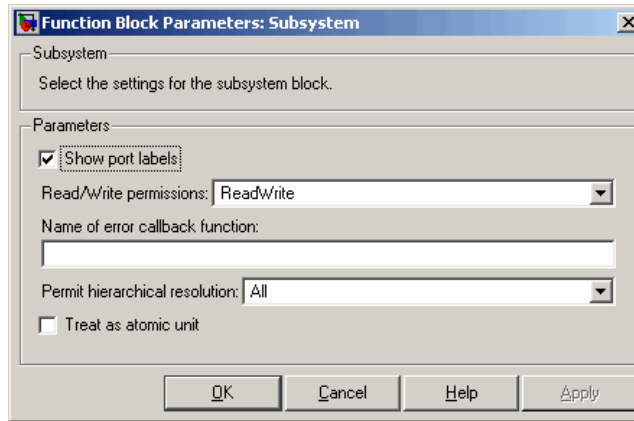
To configure your subsystem for generating modular function code, you invoke the Subsystem Parameters dialog box and make a series of selections to display and enable the **Function with separate data** option. See “Configuring Nonvirtual Subsystems for Generating Modular Function Code” on page 3-84 and “Examples of Modular Function Code for Nonvirtual Subsystems” on page 3-88 for details. For limitations that apply, see “Nonvirtual Subsystem Modular Function Code Limitations” on page 3-94.

For more information about generating code for atomic subsystems, see the sections “Nonvirtual Subsystem Code Generation” and “Generating Code and Executables from Subsystems” in the Real-Time Workshop documentation.

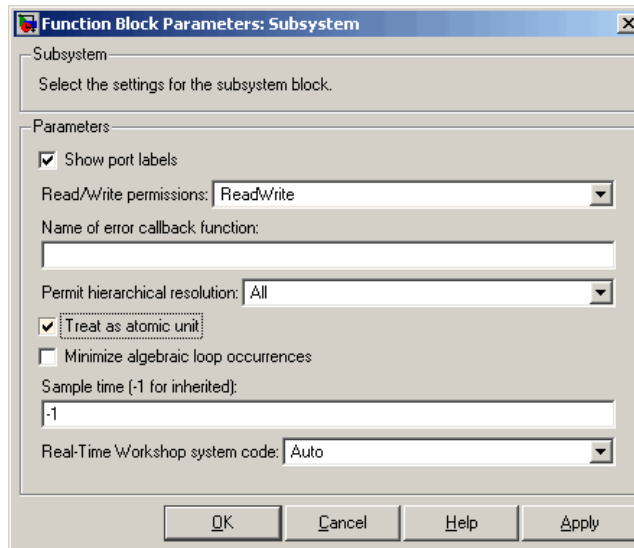
Configuring Nonvirtual Subsystems for Generating Modular Function Code

This section summarizes the steps needed to configure a subsystem in a Simulink model for modular function code generation.

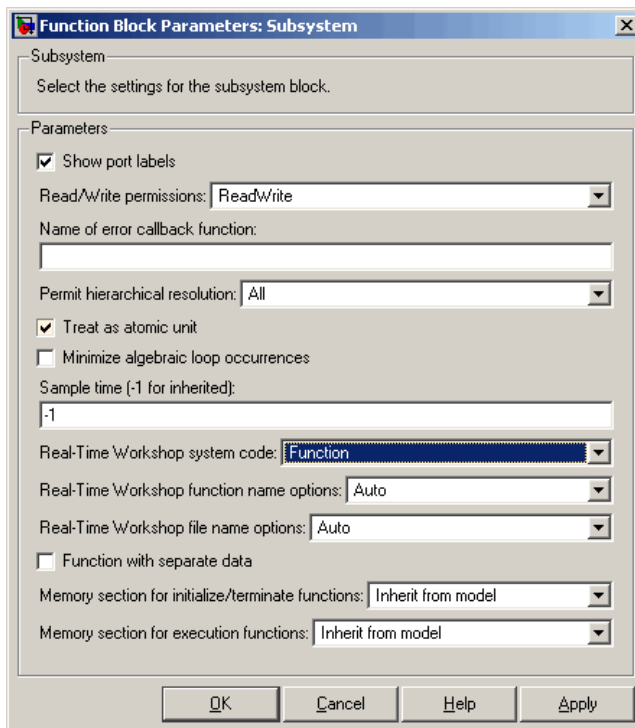
- 1 Verify that the Simulink model containing the subsystem uses an ERT-based system target file (see the **System target file** parameter on the **Real-Time Workshop** pane of the Configuration Parameters dialog box).
- 2 In your Simulink model, select the subsystem for which you want to generate modular 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.

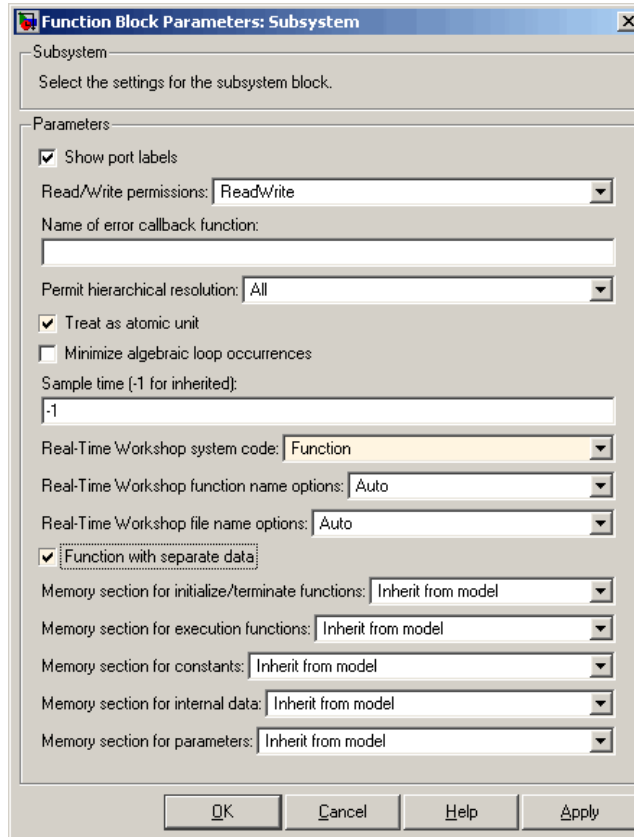


- 4** For the **Real-Time Workshop system code** parameter, select the value **Function**. After you make this selection, the **Function with separate data** option is displayed.



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**.

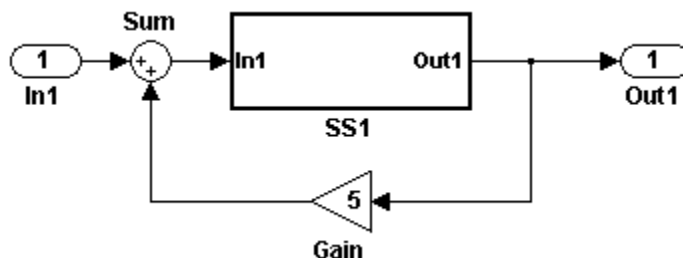
- 6 To save your subsystem parameter settings and exit the dialog box, click **OK**.

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 “Nonvirtual Subsystem Code Generation” in the Real-Time Workshop documentation. For examples of generated subsystem function code, see “Examples of Modular Function Code for Nonvirtual Subsystems” on page 3-88.

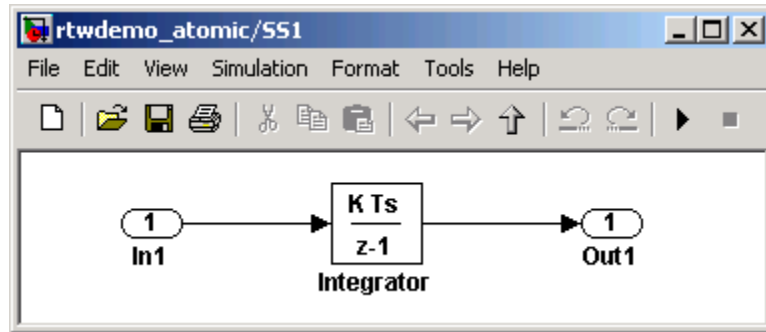
Examples of Modular Function Code for Nonvirtual Subsystems

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 a MATLAB session 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** 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`.

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 non-inlined 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 Model Function Prototypes

In this section...

“Overview” on page 3-96

“Configuring Model Function Prototypes” on page 3-97

“Model Function Prototypes Example” on page 3-105

“Configuring Model Function Prototypes Programmatically” on page 3-111

“Sample M Script for Configuring Model Function Prototypes” on page 3-114

“Verifying Generated Code for Customized Functions” on page 3-115

“Model Function Prototype Control Limitations” on page 3-115

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 3-97). 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 3-105 and the demo model `rtwdemo_fcncnprotctrl`, 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 3-111.

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 3-103.

For limitations that apply, see “Model Function Prototype Control Limitations” on page 3-115.

Configuring Model Function Prototypes

- “Launching the Model Interface Dialog Boxes” on page 3-97
- “Default Model Initialize and Step Functions View” on page 3-98
- “Model Specific C Prototypes View” on page 3-100
- “Configuring Function Prototypes for Nonvirtual Subsystems” on page 3-103

Launching the Model Interface Dialog Boxes

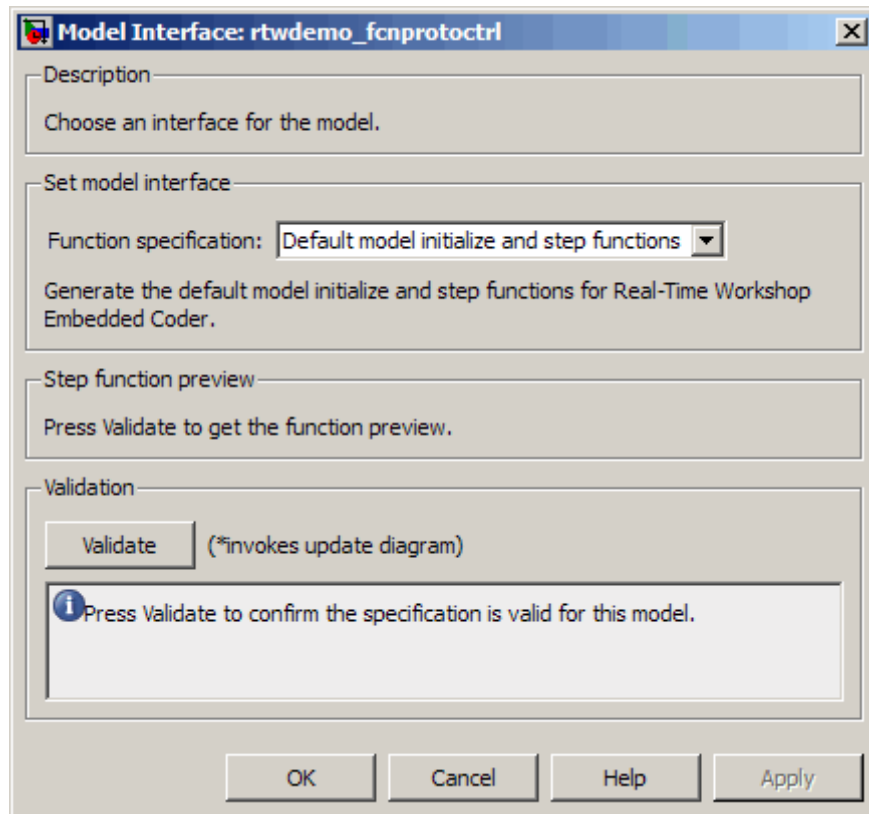
Clicking the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box launches the Model Interface

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 3-103

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: rtwdemo_fcnproctrl

Description
Choose an interface for the model.

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
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 = rtwdemo_fcnproctrl_step_custom (argIn1, * argIn2, * argOut2, * argIn3, * argIn4)

Validation
Validate (*invokes update diagram)

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)	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: <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)

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.

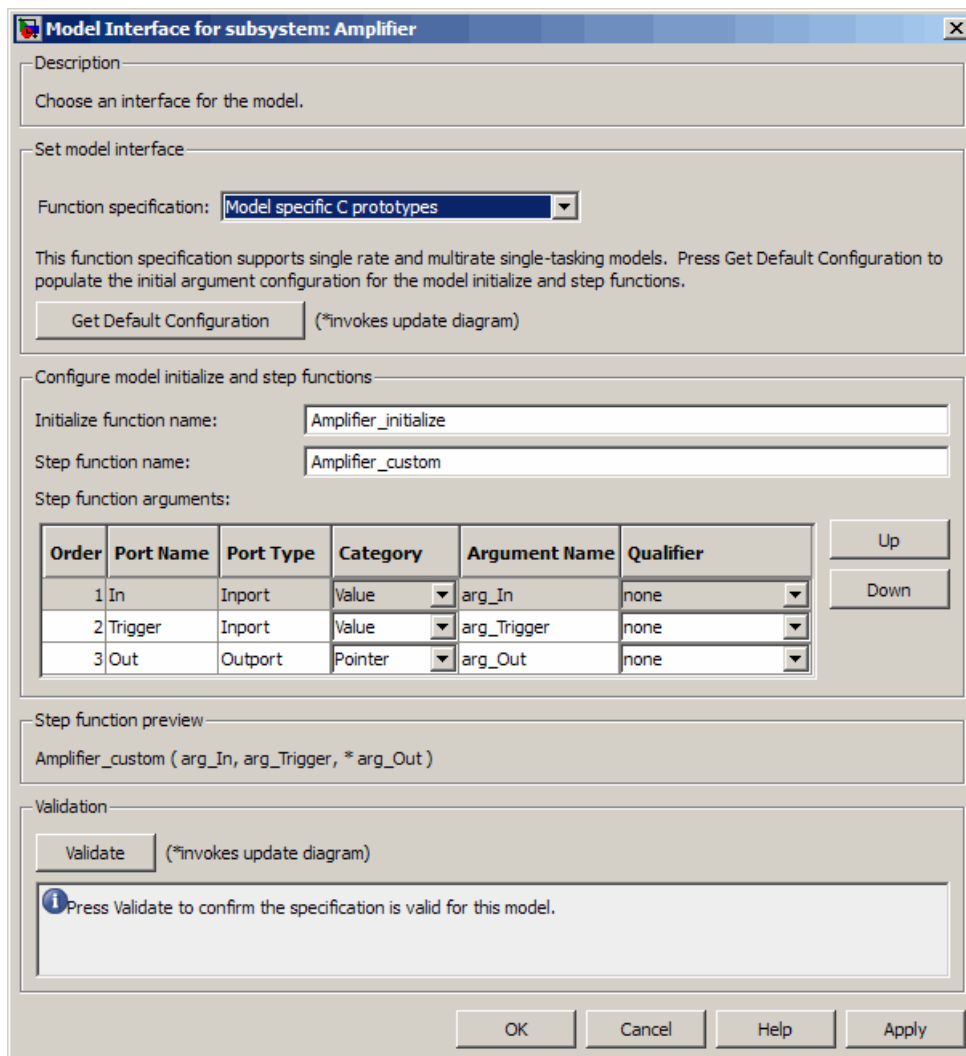
Configuring Function Prototypes for Nonvirtual Subsystems

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:



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:

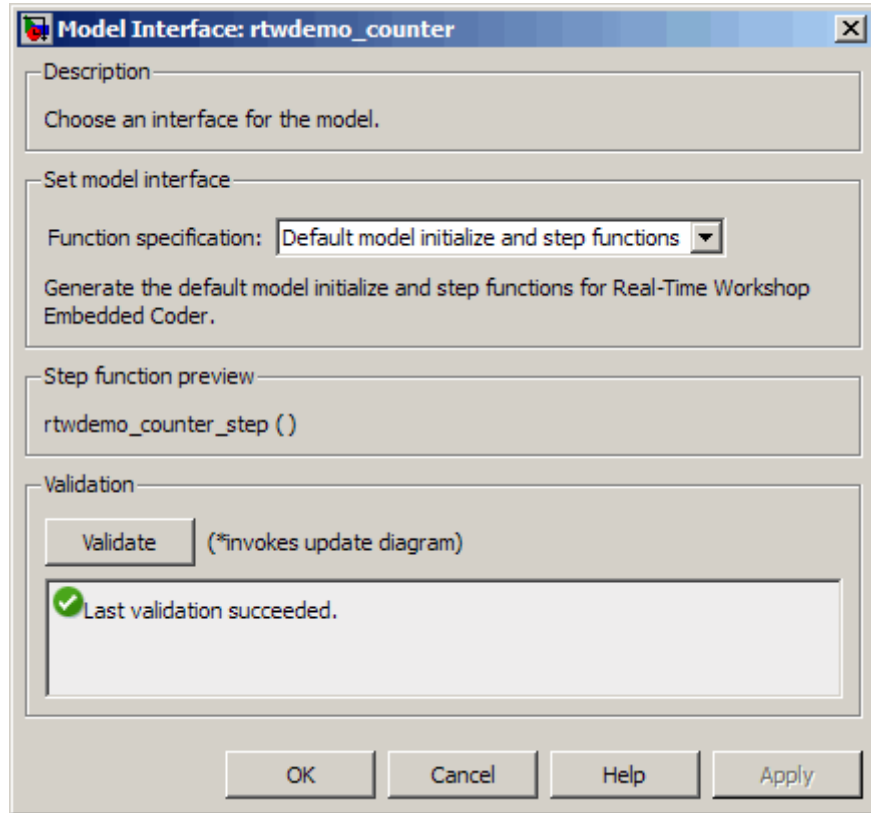
```
/* Model step function */
void rtwdemo_counter_step(void)
{
    ...
}

/* Model initialize function */
void rtwdemo_counter_initialize(void)
{
    ...
}
```

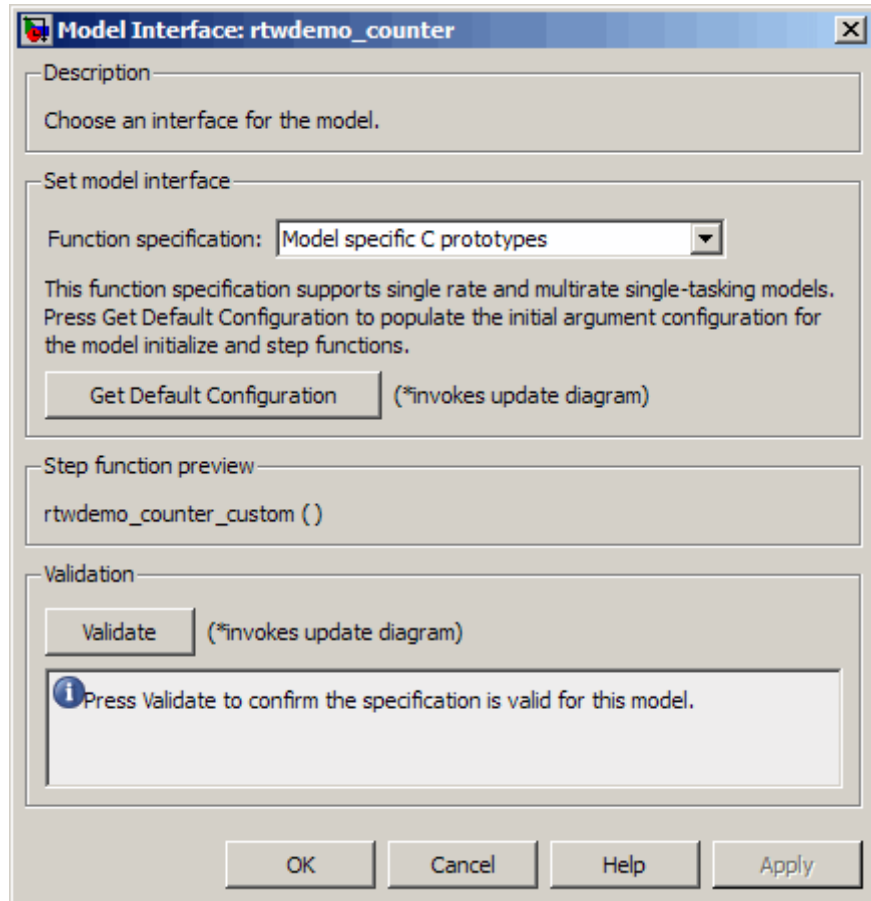
You can close the report window after you have examined the generated code. Optionally, you can save `rtwdemo_counter.c` and any other generated files of interest to a different location for later comparison.

- 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.

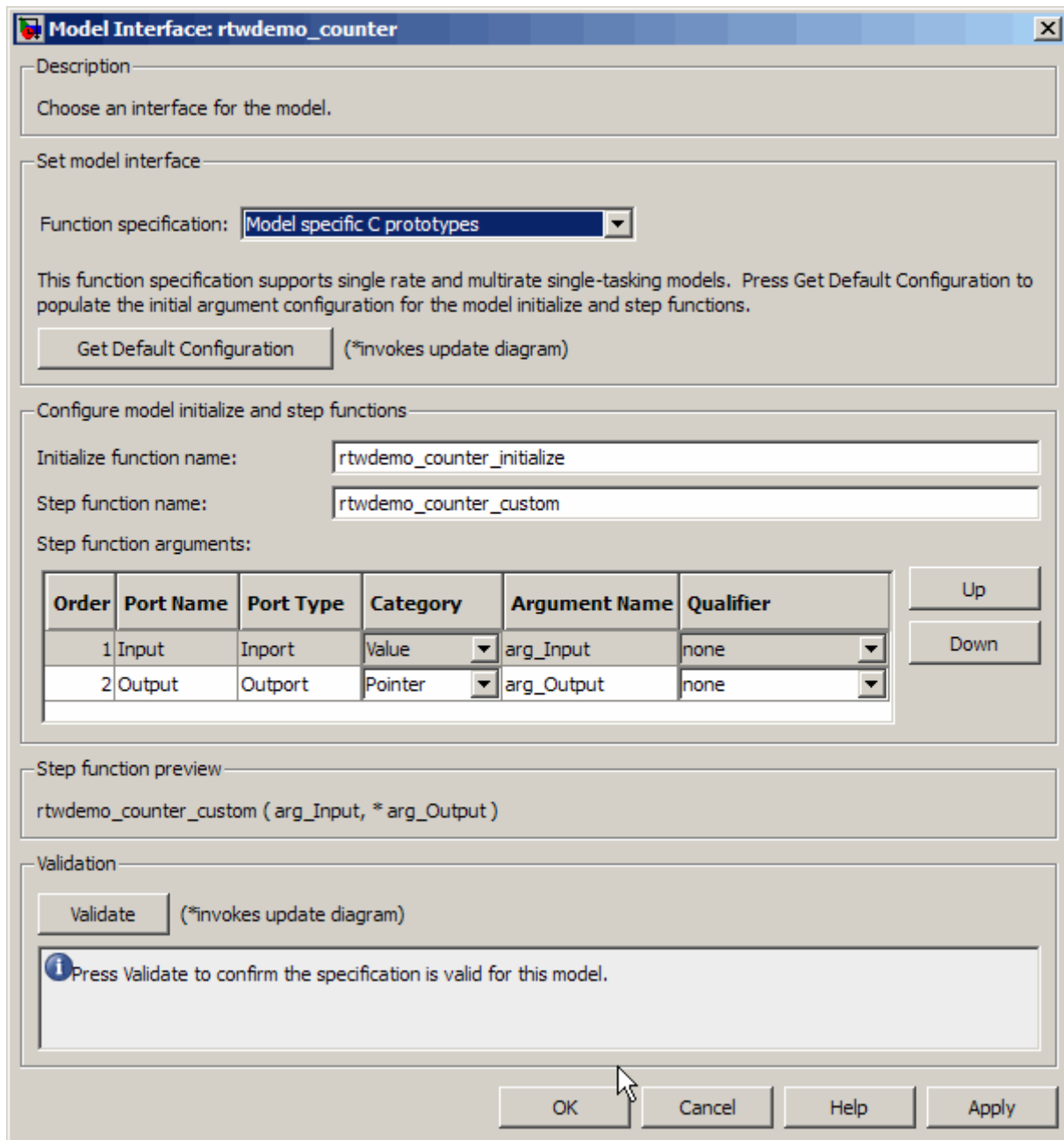
- 7 In the initial (Default model initialize and step functions) view of the Model Interface dialog box, click the **Validate** button to validate and preview the default function prototype for the `rtwdemo_counter_step` function. The function prototype arguments under **Step function preview** should correspond to the default prototype in step 4.



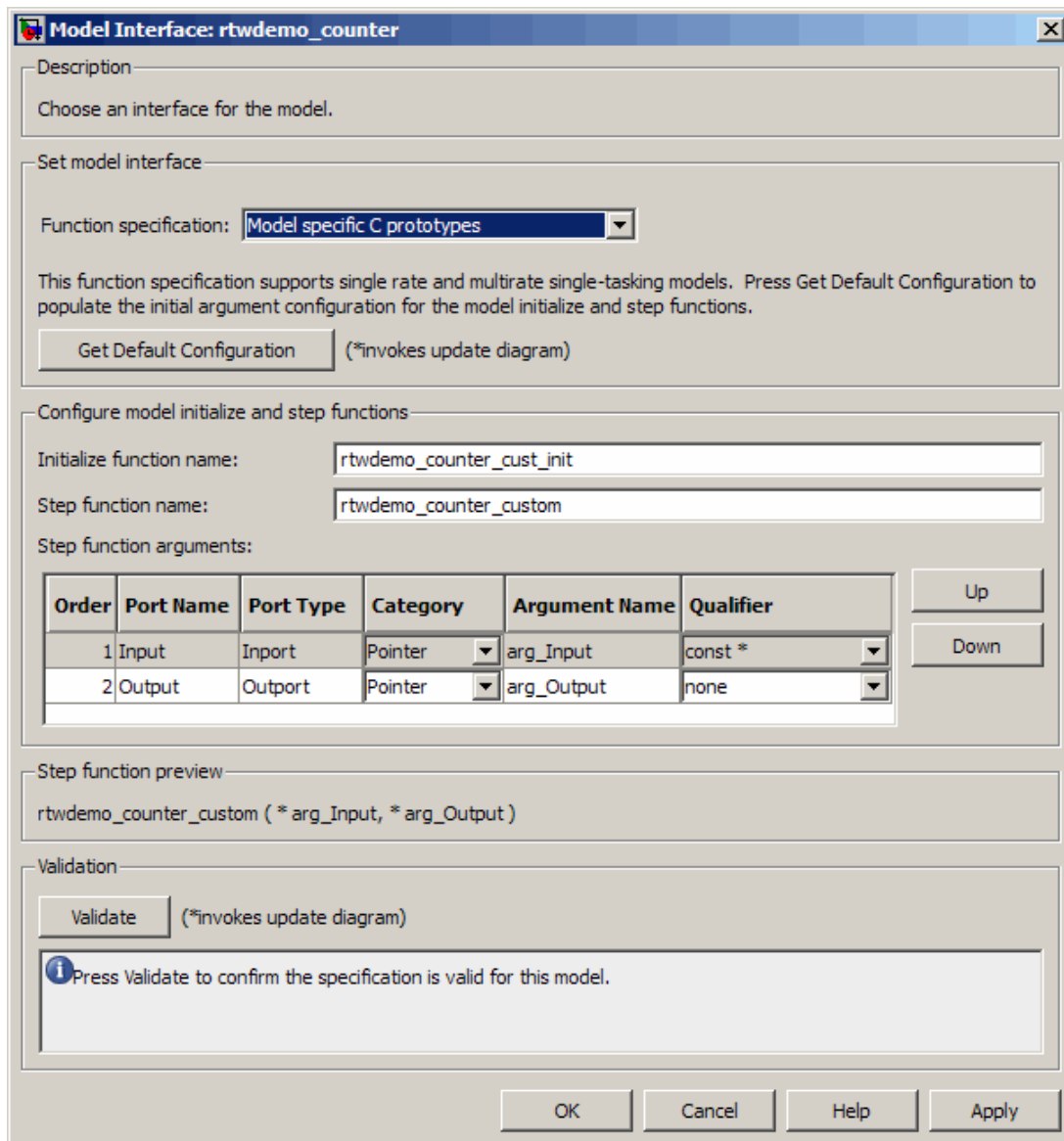
- 8 In the Model Interface dialog box, set **Function specification** field to Model specific C prototypes. Making this selection switches the dialog box from the Default model initialize and step functions view to the Model specific C prototypes view.



- 9 In the Model specific C prototypes view, click the **Get Default Configuration** button to activate the **Configure model initialize and step functions** subpane.



- 10** In the **Configure model initialize and step functions** subpane, change **Initialize function name** to `rtwdemo_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 3-113), 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 `addArgConf`.
 - 3** Attach the function prototype to your loaded ERT-based Simulink model using `attachToModel` (Function Prototype Control).

- 4** Validate the function prototype using `runValidation` (Function Prototype Control).
 - 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 3-113 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 `runValidation` (Function Prototype Control).
 - 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 `attachToModel` (Function Prototype Control).
 - 3** Get default configuration information from your model using `getDefaultConf` (Function Prototype Control).

- 4 Use the **Get** and **Set** functions listed in **Function Prototype Control Functions** on page 3-113 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 `runValidation` (**Function Prototype Control**).
- 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
<code>addArgConf</code>	Add step function argument configuration information for Simulink model port to model-specific C function prototype
<code>attachToModel</code> (Function Prototype Control)	Attach model-specific C function prototype to loaded ERT-based Simulink model
<code>getArgCategory</code> (Function Prototype Control)	Get step function argument category for Simulink model port from model-specific C function prototype
<code>getArgName</code> (Function Prototype Control)	Get step function argument name for Simulink model port from model-specific C function prototype
<code>getArgPosition</code> (Function Prototype Control)	Get step function argument position for Simulink model port from model-specific C function prototype
<code>getArgQualifier</code> (Function Prototype Control)	Get step function argument type qualifier for Simulink model port from model-specific C function prototype
<code>getDefaultConf</code> (Function Prototype Control)	Get default configuration information for model-specific C function prototype from Simulink model to which it is attached
<code>getFunctionName</code>	Get function names from model-specific C function prototype

Function Prototype Control Functions (Continued)

Function	Description
getNumArgs (Function Prototype Control)	Get number of step function arguments from model-specific C function prototype
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
runValidation (Function Prototype Control)	Validate model-specific C function prototype against Simulink model to which it is attached
setArgCategory (Function Prototype Control)	Set step function argument category for Simulink model port in model-specific C function prototype
setArgName (Function Prototype Control)	Set step function argument name for Simulink model port in model-specific C function prototype
setArgPosition (Function Prototype Control)	Set step function argument position for Simulink model port in model-specific C function prototype
setArgQualifier (Function Prototype Control)	Set step function argument type qualifier for Simulink model port in model-specific C function prototype
setFunctionName	Set function names in model-specific C function prototype

Sample M Script for Configuring Model Function Prototypes

The following sample M script configures the model function prototypes for the `rtwdemo_counter` model, using the Function Prototype Control Functions on page 3-113.

```

%% 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 generating an ERT S-function wrapper for 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 “Automatic S-Function Wrapper Generation” on page 3-68 and “Verifying Generated Code with Software-in-the-Loop” on page 4-2.

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 **Override optimizations and always copy signal** check box in the Signal Conversion block parameters.

Generating and Controlling C++ Encapsulation Interfaces

In this section...

“Overview of C++ Encapsulation” on page 3-118

“C++ Encapsulation Quick-Start Example” on page 3-119

“Generating and Configuring C++ Encapsulation Interfaces to Model Code” on page 3-126

“Configuring C++ Encapsulation Interfaces Programmatically” on page 3-133

“Sample M-Script for Configuring the Step Method for a Model Class” on page 3-136

“C++ Encapsulation Interface Control Limitations” on page 3-138

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 3-132.)

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 3-119. For more details about configuring C++ encapsulation interfaces for your model code, see “Generating and Configuring C++ Encapsulation Interfaces to Model Code” on page 3-126 and “Configuring C++ Encapsulation Interfaces Programmatically” on page 3-133. For limitations that apply, see “C++ Encapsulation Interface Control Limitations” on page 3-138.

Note For a demonstration of the C++ encapsulation capability, see the demo model `rtwdemo_cppencap`.

C++ Encapsulation Quick-Start Example

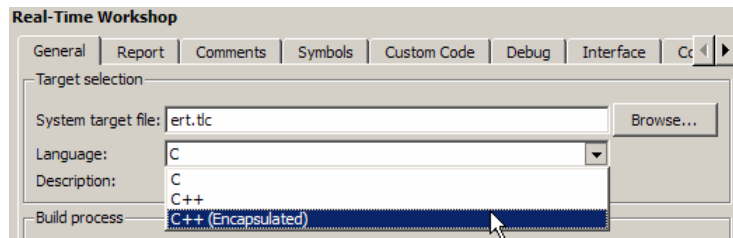
This example illustrates a simple use of the C++ (Encapsulated) option. It uses C++ encapsulation to generate interfaces for code from a Real-Time Workshop demo model, without extensive modifications to default settings.

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 3-126.

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**.

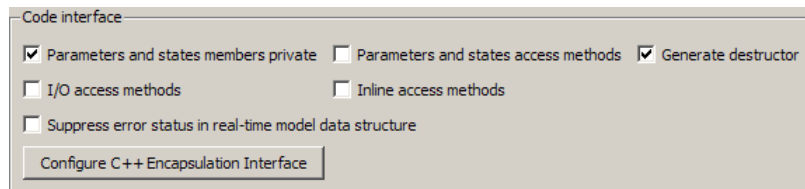
- Optionally, as a baseline for later code comparison, generate code from the model using a different **Language** parameter setting, C++ or C. (You can set up the build directory naming or location to distinguish your baseline build from later builds of the same model.)
- On the **Real-Time Workshop** pane of the Configuration Parameters dialog box, select the C++ (Encapsulated) language option.



Click **Apply**.

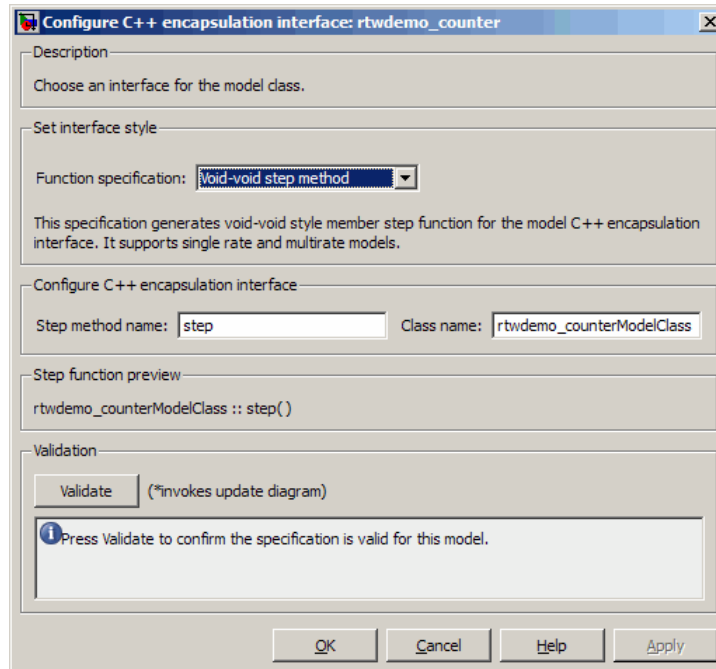
Note To immediately generate the default style of encapsulated C++ code, without exploring the related model configuration options, skip steps 5–9 and go directly to step 10.

- 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 3-126 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.

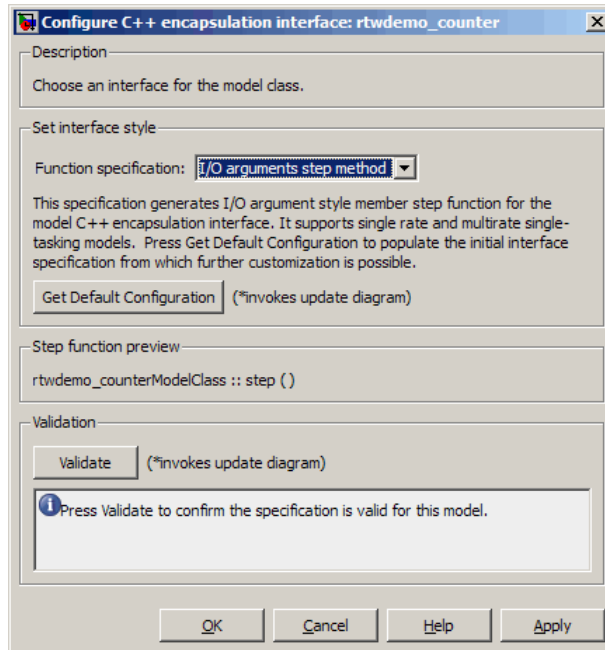


See “Configuring the Step Method for Your Model Class” on page 3-128 for descriptions of these controls.

Note If the void-void interface style meets your needs, you can skip steps 7–9 and go directly to step 10.

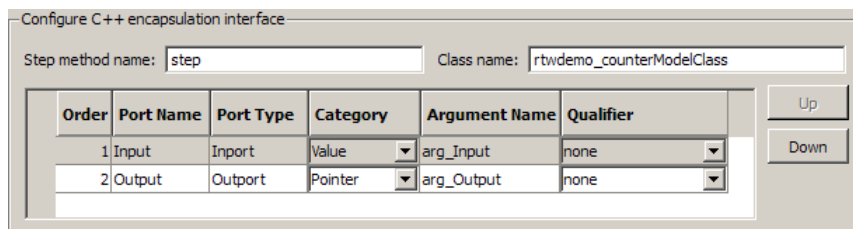
- 7** If you want root-level model input and output to be arguments on the step method, select the value I/O arguments step method from the **Function**

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 3-128 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 3-130 for descriptions of these controls.

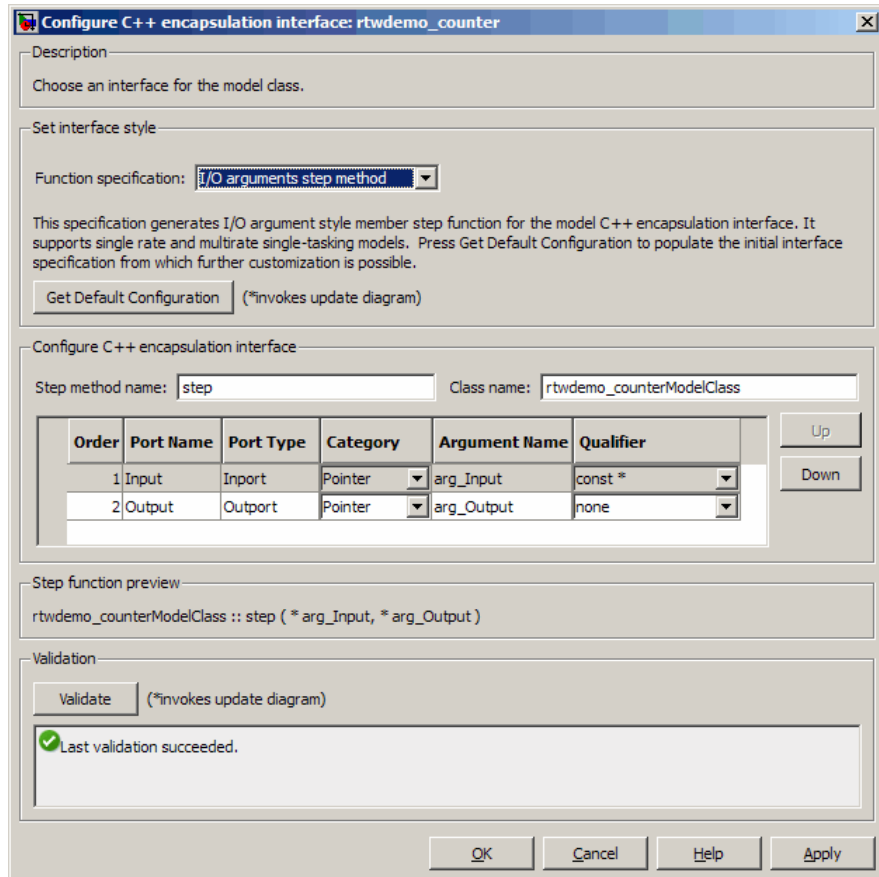
- 9 Perform this optional step only if you want to customize the configuration of the I/O arguments generated for your model step method.

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 "Contents" panel with a tree view containing the following items: Summary, Subsystem Report, Traceability Report, Generated Source Files, ert_main.cpp, rtwdemo_counter.cpp, rtwdemo_counter.h, rtwdemo_counter_private.h, rtwdemo_counter_types.h, and rtwtypes.h. The main area of the window displays C++ code for the `rtwdemo_counterModelClass` class. The code includes comments and function declarations for `initialize`, `step`, `terminate`, a constructor, and a destructor. It also lists private data members such as `BlockIO`, `DWork`, `PrevZCSigStates`, `RT_MODEL`, and `RT_MODEL` pointers.

```

51  /* Class declaration for model rtwdemo_counter */
52  class rtwdemo_counterModelClass {
53  /* public data and function members */
54  public:
55  /* Model entry point functions */
56
57  /* model initialize function */
58  void initialize(void);
59
60  /* model step function */
61  void step(const int32_T *arg_Input, int32_T *arg_Output);
62
63  /* model terminate function */
64  void terminate(void);
65
66  /* Constructor */
67  rtwdemo_counterModelClass(void);
68
69  /* Destructor */
70  ~rtwdemo_counterModelClass();
71  RT_MODEL_rtwdemo_counter * getRTM(void);
72
73  /* private data and function members */
74  private:
75  BlockIO_rtwdemo_counter rtwdemo_counter_B; /* Block signals */
76  D_Work_rtwdemo_counter rtwdemo_counter_DWork; /* Block states */
77  PrevZCSigStates_rtwdemo_counter rtwdemo_counter_PrevZCSigState; /* Trigg
78  RT_MODEL_rtwdemo_counter rtwdemo_counter_M; /* Real-time model */
79  RT_MODEL_rtwdemo_counter *rtwdemo_counter_M; /* Real-time model pointer
80  };

```

Note

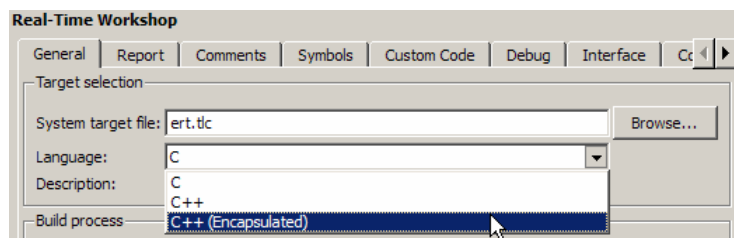
- If you configured custom I/O arguments for the model step method (optional step 9), examine the generated code for the step method in `rtwdemo_counter.h` and `rtwdemo_counter.cpp`. The arguments should reflect your changes. For example, if you performed the Input argument modifications in step 9, the 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

- “Selecting the C++ (Encapsulated) Option” on page 3-126
- “Configuring Code Interface Options” on page 3-126
- “Configuring the Step Method for Your Model Class” on page 3-128
- “Configuring C++ Encapsulation Interfaces for Nonvirtual Subsystems” on page 3-132

Selecting the C++ (Encapsulated) Option

To select the C++ (Encapsulated) option, use the **Language** menu on the **Real-Time Workshop** pane of the Configuration Parameters dialog box or Model Explorer:

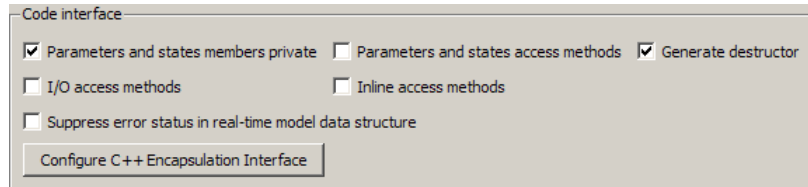


Selecting this option has the following effects on other panes in the Configuration Parameters dialog box:

- Disables model configuration options that C++ (Encapsulated) does not support. For details, see “C++ Encapsulation Interface Control Limitations” on page 3-138.
- Replaces the default options in the **Code interface** subpane on the **Interface** pane 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 default options in the **Code interface** subpane on the **Interface** pane.



- **Parameters and states members private**

Generates non-I/O model structures, including states and parameters, as private data members (on by default).

- **Parameters and states access methods**

Generates get/set access methods for non-I/O model structures, including states and parameters (off by default).

- **Generate destructor**

Generates a destructor for the model class (on by default).

- **I/O access methods**

Generates access methods for root-level I/O signals if possible (off by default).

Note This option affects generated code only if you are using the default (void-void style) step method for your model class, and *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 3-128 and “Passing I/O Arguments” on page 3-130.

- **Inline access methods**

Inlines generated access methods (off by default).

- **Suppress error status in real-time model data structure**

Omits the error status field from the generated real-time model data structure rtModel (off by default). This option reduces memory usage.

Selecting this option can cause the code generator to omit the `rtModel` data structure from generated code.

- **Configure C++ Encapsulation Interface**

Launches 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 3-128.

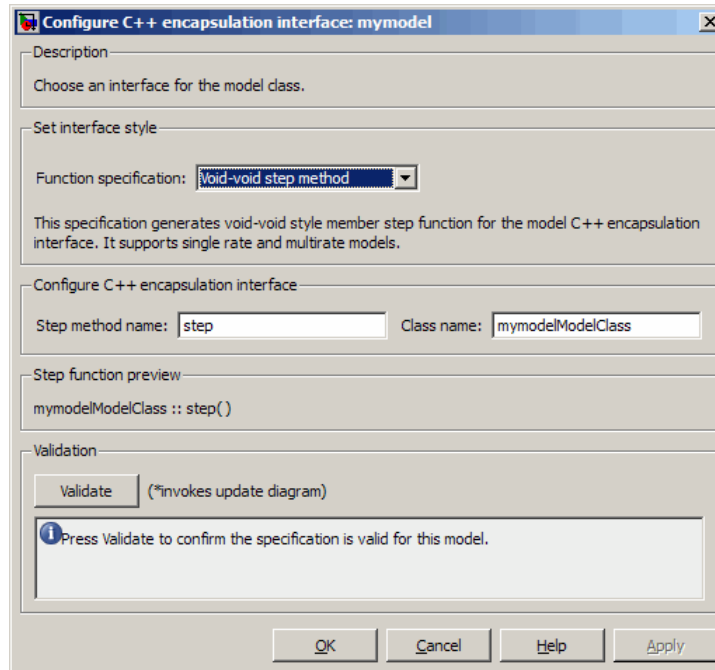
Configuring the Step Method for Your Model Class

To configure the step method for your model class, click the **Configure C++ Encapsulation Interface** button, which appears on the **Interface** pane when you select **C++ (Encapsulated)** for your model. This action opens the Configure C++ encapsulation interface dialog box, which allows you to configure the step method for your model class in either of two styles:

- “Passing No Arguments (void-void)” on page 3-128
- “Passing I/O Arguments” on page 3-130

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**

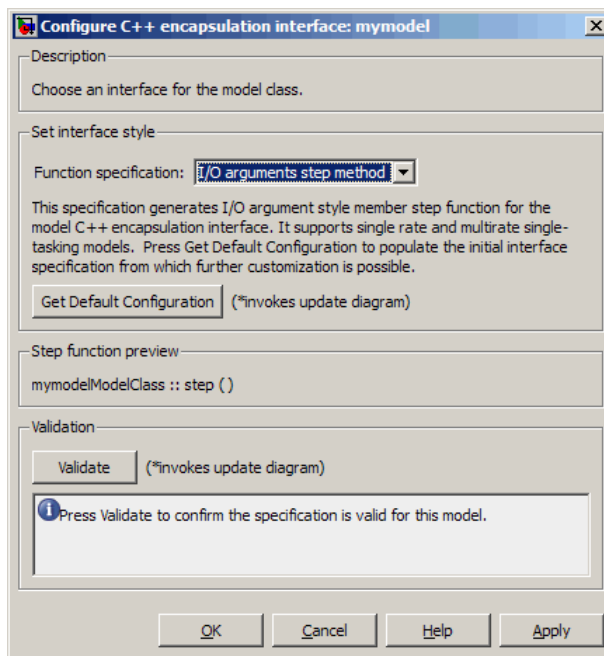
Displays a preview of the model step function prototype as currently configured. The preview display is dynamically updated as you make configuration changes.

- **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:

Configure C++ encapsulation interface

Step method name: Class name:

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

Up
Down

- **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)

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 3-128. (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 3-135) 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` (C++ Encapsulation Interface Control).
 - 3** Get default C++ encapsulation interface configuration information from your model using `getDefaultConf` (C++ Encapsulation Interface Control).
 - 4** Use the `Get` and `Set` functions listed in C++ Encapsulation Interface Control Functions on page 3-135 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` (C++ Encapsulation Interface Control). (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 3-135 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` (C++ Encapsulation Interface Control). (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
attachToModel (C++ Encapsulation Interface Control)	Attach model-specific C++ encapsulation interface to loaded ERT-based Simulink model
getArgCategory (C++ Encapsulation Interface Control)	Get argument category for Simulink model port from model-specific C++ encapsulation interface
getArgName (C++ Encapsulation Interface Control)	Get argument name for Simulink model port from model-specific C++ encapsulation interface
getArgPosition (C++ Encapsulation Interface Control)	Get argument position for Simulink model port from model-specific C++ encapsulation interface
getArgQualifier (C++ Encapsulation Interface Control)	Get argument type qualifier for Simulink model port from model-specific C++ encapsulation interface
getClassName	Get class name from model-specific C++ encapsulation interface
getDefaultConf (C++ Encapsulation Interface Control)	Get default configuration information for model-specific C++ encapsulation interface from Simulink model to which it is attached
getNumArgs (C++ Encapsulation Interface Control)	Get number of step method arguments from model-specific C++ encapsulation interface
getStepMethodName	Get step method name from model-specific C++ encapsulation interface

C++ Encapsulation Interface Control Functions (Continued)

Function	Description
<code>RTW.configSubsystemBuild</code>	Open GUI to configure C function prototype or C++ encapsulation interface for right-click build of specified subsystem
<code>RTW.getEncapsulation-InterfaceSpecification</code>	Get handle to model-specific C++ encapsulation interface control object
<code>runValidation (C++ Encapsulation Interface Control)</code>	Validate model-specific C++ encapsulation interface against Simulink model to which it is attached
<code>setArgCategory (C++ Encapsulation Interface Control)</code>	Set argument category for Simulink model port in model-specific C++ encapsulation interface
<code>setArgName (C++ Encapsulation Interface Control)</code>	Set argument name for Simulink model port in model-specific C++ encapsulation interface
<code>setArgPosition (C++ Encapsulation Interface Control)</code>	Set argument position for Simulink model port in model-specific C++ encapsulation interface
<code>setArgQualifier (C++ Encapsulation Interface Control)</code>	Set argument type qualifier for Simulink model port in model-specific C++ encapsulation interface
<code>setClassName</code>	Set class name in model-specific C++ encapsulation interface
<code>setStepMethodName</code>	Set step method name in model-specific C++ encapsulation interface

Sample M-Script for Configuring the Step Method for a Model Class

The following sample M-script configures the step method for the `rtwdemo_counter` model class, using the C++ Encapsulation Interface Control Functions on page 3-135.

```
%% 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.
- A generated model class provides an encapsulated interface only to the top model in a model reference hierarchy. The C++ (Encapsulated) **Language** setting should be selected only for the top model in a hierarchy, not for the referenced models. If you build a model containing a Model block whose referenced model has C++ (Encapsulated) selected as the **Language** setting, the build reports an error.

- 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 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 **Override optimizations and always copy signal** check box in the Signal Conversion block parameters.

Creating and Using Host-Based Shared Libraries

In this section...
“Overview” on page 3-140
“Generating a Shared Library Version of Your Model Code” on page 3-141
“Creating Application Code to Load and Use Your Shared Library File” on page 3-142
“Host-Based Shared Library Limitations” on page 3-146

Overview

The Real-Time Workshop Embedded Coder software 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 Microsoft Windows dynamic link library (.dll) file or a UNIX³ 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 platforms, use the Dependency Walker utility, downloadable from <http://www.dependencywalker.com>
- On UNIX platforms, use `nm -D model.so`

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

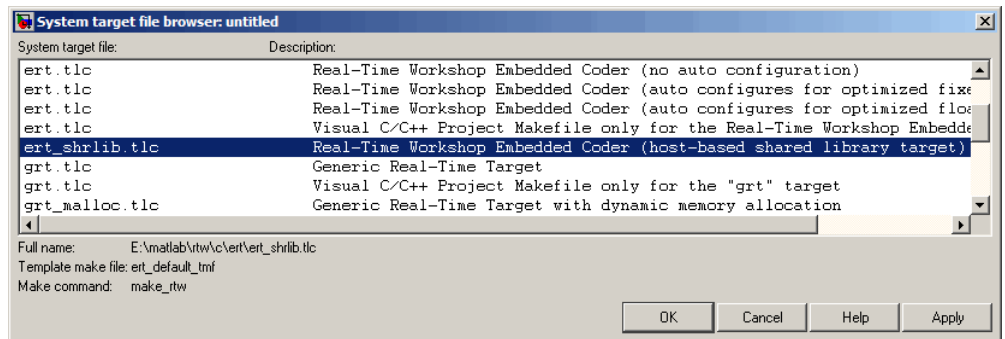
To generate and use a host-based shared library, you

- 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 directory. 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 subdirectory, and the `.dll` file or `.so` file that has been generated into your current directory.

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_shrllib`. Clicking the blue button in the demo model runs a script that:

- 1 Builds a shared library file from the model (for example, `rtwdemo_shrllib_win32.dll` on 32-bit Windows platforms)
- 2 Compiles and links an example application, `rtwdemo_shrllib_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 directory before running the `rtwdemo_shrllib` script.

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

File	Description
<code>rtwdemo_shrllib_app.h</code>	Example application header file
<code>rtwdemo_shrllib_app.c</code>	Example application that loads and uses the shared library file generated for the demo model
<code>run_rtwdemo_shrllib_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 directory. 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;
```

```
...
#ifdef _WIN64
    handleLib = LOADLIB("./rtwdemo_shrplib_win64.dll");
#else
#ifdef _WIN32
    handleLib = LOADLIB("./rtwdemo_shrplib_win32.dll");
#else /* UNIX */
    handleLib = LOADLIB("./rtwdemo_shrplib.so", RTLD_LAZY);
#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_shrplib (*mdl_Uptr);
ExternalOutputs_rtwdemo_shrplib (*mdl_Yptr);

uint8_T (*sum_outptr);
...
#ifdef LCCDLL || defined(BORLANDCDLL)
/* Exported symbols contain leading underscores when DLL is linked with
LCC or BORLANDC */
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");
#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 M 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 the Microsoft® Visual C++® development system, 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.

Verifying Generated Code

- “Verifying Generated Code with Software-in-the-Loop” on page 4-2
- “Verifying Generated Code with Processor-in-the-Loop” on page 4-7
- “Types of PIL Functionality” on page 4-10
- “Configuring PIL Mode in the Model Block” on page 4-22
- “Creating a Connectivity Configuration for Your Target” on page 4-26
- “PIL Feature Support and Limitations” on page 4-31

Verifying Generated Code with Software-in-the-Loop

In this section...

“Overview” on page 4-2

“Validating Generated Code on the MATLAB Host Computer Using Hardware Emulation” on page 4-3

“Validating ERT Production Code on the MATLAB Host Computer Using Portable Word Sizes” on page 4-4

Overview

Real-Time Workshop Embedded Coder software provides software-in-the-loop (SIL) code verification for subsystems using ERT S-function wrappers, described in “Automatic S-Function Wrapper Generation” on page 3-68. When processor word sizes differ between host and target platforms (for example, a 32-bit host and a 16-bit target), there are two ways to configure your Simulink model to simulate target behavior on the MATLAB host with SIL:

- Enable and select **Emulation hardware** settings on the **Hardware Implementation** pane of the Configuration Parameters dialog
- Select the **Enable portable word sizes** option on the **Interface** pane of the Configuration Parameters dialog

Select the *hardware emulation* method if you need to guarantee bit-true agreement for integer and fixed-point operations between the results of simulations run on the MATLAB host computer and the results of generated code executed on the target deployment system. In this case, the code that you generate for simulation on the MATLAB host might contain additional code, such as data type casts, that is necessary to ensure behavior consistent with the target environment.

See also “Configuring Optimizations” in the Real-Time Workshop documentation for settings in the **Code generation** subpane of the **Optimization** pane that affect the generated code. After SIL testing on the MATLAB host, you must select **None** for **Emulation hardware** and then regenerate code for the target before deployment.

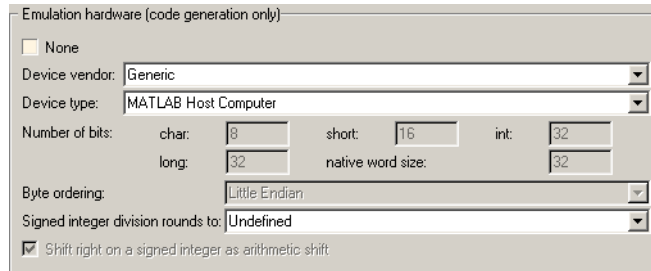
Select the *portable word sizes* method if you want to generate code that can be compiled without alteration both for SIL testing on the MATLAB host computer and deployment on the target system. In this case, the code that you generate has conditional processing macros that allow you to first compile for the host platform, using the compiler option `-DPORTABLE_WORDSIZES`, and then compile for the target platform, omitting the option.

To illustrate both methods of configuring your model to simulate target behavior on the MATLAB host, The MathWorks provides the demo model `rtwdemo_sil`. The demo allows you to simulate the same model using each method, and to compare model configuration settings and results.

Validating Generated Code on the MATLAB Host Computer Using Hardware Emulation

Real-Time Workshop Embedded Coder software provides **Emulation hardware** settings that support code generation for host-target configurations in which the processor word sizes differ between host and target platforms (for example, a 32-bit host and a 16-bit target). Selecting **MATLAB Host Computer** as the **Emulation hardware** device type allows you to generate model code with any additional code, such as data type casts, that is necessary to ensure behavior on the MATLAB host computer that is consistent with the target environment.

To use this feature, go to the **Emulation hardware** subpane of the **Hardware Implementation** pane of the Configuration Parameters dialog box, clear the **None** option if it is selected, select **Generic** as the **Device vendor** if it is not already selected, and select **MATLAB Host Computer** as the **Device type**. Also, go to the **Interface** pane, select **Create Simulink (S-Function) block**, and make sure that **Enable portable word sizes** is cleared.



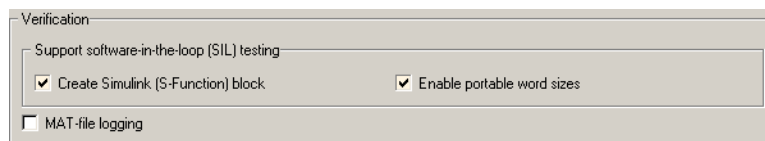
You can then right-click the subsystem that you want to test on the MATLAB host, and select **Real-Time Workshop > Build Subsystem** to build it. This will generate an S-function wrapper for the generated subsystem code, which can be used on the host to verify that the generated code provides the same result as the original subsystem.

For an example of SIL testing using hardware emulation, see `rtwdemo_sil`.

Validating ERT Production Code on the MATLAB Host Computer Using Portable Word Sizes

The Real-Time Workshop Embedded Coder software provides a model configuration option, **Enable portable word sizes**, that supports code generation for host-target configurations in which the processor word sizes differ between host and target platforms (for example, a 32-bit host and a 16-bit target). Selecting the **Enable portable word sizes** option allows you to generate code with conditional processing macros that allow the same generated source code files to be used both for SIL testing on the host platform and for production deployment on the target platform.

To use this feature, select both **Create Simulink (S-Function) block** and **Enable portable word sizes** on the **Interface** pane of the Configuration Parameters dialog box. Also, make sure that **Emulation hardware** is set to **None** on the **Hardware Implementation** pane.



When you generate code from your model, data type definitions are conditionalized such that `tmwtypes.h` is included to support SIL testing on the host platform and Real-Time Workshop types are used to support deployment on the target platform. For example, in the generated code below, the first two lines define types for host-based SIL testing and 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 */

```

To build the generated code for SIL testing on the host platform, the definition `PORTABLE_WORDSIZES` should be passed to the compiler, for example by using the compiler option `-DPORTABLE_WORDSIZES`. To build the same code for target deployment, the code should be compiled without the `PORTABLE_WORDSIZES` definition.

For an example of SIL testing using portable word sizes, see `rtwdemo_sil`.

Portable Word Sizes Limitations

The following limitations apply to performing SIL testing using the **Enable portable word sizes** model configuration parameter.

- Numerical results of the S-function simulation on the MATLAB host may differ from results on the actual target 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

Verifying Generated Code with Processor-in-the-Loop

In this section...

“What Is Processor-in-the-Loop Cosimulation?” on page 4-7

“Comparison of PIL and SIL” on page 4-7

“Why Use PIL Cosimulation?” on page 4-8

“Problems That PIL Can Detect” on page 4-8

“How PIL Cosimulation Works” on page 4-9

What Is Processor-in-the-Loop Cosimulation?

You can use processor-in-the-loop (PIL) cosimulation to verify your generated code. PIL is a technique designed to help you evaluate the behavior of a candidate algorithm (e.g., a control or signal processing algorithm) on the target processor (or an instruction set simulator) selected for the application. In PIL cosimulation, the target processor participates fully in the simulation loop—hence the term *processor-in-the-loop cosimulation*. You can compare the output of regular simulation modes (e.g., Normal or Accelerator) and PIL cosimulation mode to check that your generated code performs as intended.

You can easily switch between simulation 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.

Comparison of PIL and SIL

Software-in-the-loop (SIL) allows you to simulate target behavior on the host machine. SIL tests the generated code via an S-function wrapper on the host computer. Execution is host/host and non-real-time.

Processor-in-the-loop (PIL) allows you to test the generated code as cross-compiled object code on the target processor (or equivalent instruction

set simulator). PIL exercises the same object code that will be used in production software. Execution is host/target and non-real-time.

Why Use PIL Cosimulation?

Normal simulation techniques do not help you to account for restrictions and requirements imposed by the hardware (e.g., limited memory resources, or behavior of target-specific optimized code). PIL cosimulation 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 in the context of the original model. You can avoid any divergence between the deployed system and your original model. If you need to return to the original model and change it, you can achieve faster iteration between model development and generated code verification.

Problems That PIL Can Detect

PIL is useful for detecting problems such as:

- Code generator bugs
- Compiler bugs
- Incorrect target-specific code
 - Undetectable with software-in-the-loop (SIL) 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 give slightly different results in simulation and on hardware owing 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.

You can also use PIL simulation with a debugger, which allows access to debugger features such as code coverage reports and execution profiling.

How PIL Cosimulation Works

In a PIL simulation, Real-Time Workshop Embedded Coder software generates code for the part of the model that you have designated to simulate in PIL mode. This code is cross-compiled for the target hardware and runs on the target platform. A part of the model remains in Simulink without the use of code generation. Typically this part of the model includes test stimulus signals or an environment model to stimulate the component running in PIL mode.

During PIL simulation, Simulink simulates the non-PIL part of the model for one sample interval and sends the output signals to the target platform. When the target platform receives signals from Simulink, it executes the PIL algorithm for one sample step. The PIL algorithm returns its output signals computed during this step to Simulink. At this point, one sample cycle of the simulation is complete and the Simulink model 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, the Simulink test harness and the object code exchange *all* I/O data.

For more information on different types of PIL functionality, see the next section, “Types of PIL Functionality” on page 4-10.

For instructions and demos of using PIL with Real-Time Workshop Embedded Coder software, see

- “Configuring PIL Mode in the Model Block” on page 4-22
- “Creating a Connectivity Configuration for Your Target” on page 4-26

Types of PIL Functionality

In this section...

“PIL Mode in the Model Block ” on page 4-10

“PIL Behavior and Model Referencing” on page 4-11

“PIL Block For Embedded IDE Link Products” on page 4-12

“Code Interfaces For PIL” on page 4-13

“Comparison of the Model Block in PIL Mode and the PIL Block” on page 4-15

“PIL Modeling Scenarios with the Model Block” on page 4-15

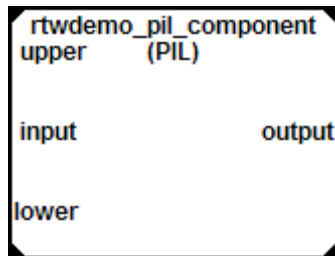
“Modeling Scenarios with the PIL Block” on page 4-20

PIL Mode in the Model Block

Processor-in-the-loop (PIL) mode is available in the Model block only as a Real-Time Workshop Embedded Coder feature.

You can switch the Model block between simulation and PIL modes. This allows you to easily verify the generated code by executing the referenced model as compiled code in the target environment. Your embedded software component can be modelled and tested 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 the production hardware.

When a Model block is in PIL mode, the label (PIL) appears on the block.



With Model block PIL mode, you can use the Processor-in-the-Loop (PIL) Connectivity API to apply the power of PIL verification to your target processor. With so many different target environments, there are countless tools and approaches for building, downloading, and communicating with an executable. You can use the API to integrate third-party tools for:

- Building the PIL application (i.e., 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 allow the required level of automation. For hardware cases that The MathWorks does not support, see “PIL Feature Support and Limitations” on page 4-31.

See the following overview sections to understand how PIL works in the Model block:

- “PIL Behavior and Model Referencing” on page 4-11
- “Code Interfaces For PIL” on page 4-13
- “PIL Modeling Scenarios with the Model Block” on page 4-15

See the following sections for instructions and demos on using Model block PIL mode and the Target Connectivity API:

- “Using PIL Mode in the Model Block” on page 4-22
- “Creating a Connectivity Configuration for Your Target” on page 4-26
- “Demos of the Target Connectivity API” on page 4-29

For an introduction to the Model block, see the Model block section in the Simulink reference documentation.

PIL Behavior and Model Referencing

You can view your model hierarchy in the Model Dependency Viewer.

In the Referenced Model Instances view, Model blocks appear differently to indicate Normal, Accelerator, 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 against the simulation targets (or SIM targets) of the blocks beneath. See “Simulation Targets” in the Simulink documentation.
- A Model block in PIL mode overrides the behavior of any Model blocks beneath it in the model reference hierarchy. The PIL mode Model block uses the model reference targets of the blocks beneath. See “Code Interface for PIL Mode in the Model Block” on page 4-14.

For a block to behave as specified by its simulation mode, it must have a Normal mode path from the top-level model.

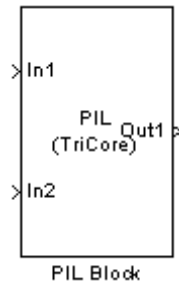
Note Only one branch in a Model reference hierarchy can simulate in PIL mode.

For an example model hierarchy, see “PIL Modeling Scenarios with the Model Block” on page 4-15.

PIL Block For Embedded IDE Link Products

The PIL block is available only with Embedded IDE Link products. You cannot use the Target Connectivity API.

With an Embedded IDE Link product, you can automatically create a PIL block to test the code generated from your model.



See the following sections to understand the differences between Model block PIL and the PIL block:

- “Code Interfaces For PIL” on page 4-13
- “Comparison of the Model Block in PIL Mode and the PIL Block” on page 4-15
- “Modeling Scenarios with the PIL Block” on page 4-20
- “PIL Feature Support and Limitations” on page 4-31

Code Interfaces For PIL

To understand the difference between Model block PIL mode (with Real-Time Workshop Embedded Coder) and the PIL block (with Embedded IDE Link products), it is helpful to first understand the different code interfaces that the code generation products produce.

“Standalone code” is generated when you perform a top-level or right-click subsystem build for a single deployable component. It can be compiled and linked into a standalone executable or integrated with other code. For more information on the standalone code interface, see “Model Entry Points” on page 2-24.

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 model reference targets as needed to compute the referenced model outputs. Model reference targets are sometimes called Real-Time Workshop targets. For more information,

see “Generating Code for Model Referencing” in the Real-Time Workshop documentation.

Note The model reference target does not have the same code interface as standalone code.

If you want to integrate automatically generated code with other code, you should use standalone code, because the standalone code interface (e.g. entry points) is fully documented.

Code Interface for PIL Mode in the Model Block

When you run a simulation with a Model block in PIL mode, the software calls the model reference target for the Model block (if it already exists), or generates the model reference target if it does not yet exist.

There are three ways you can generate the model reference target (if it does not yet exist):

- Run the simulation.
- Build the top model containing the Model block (press **Ctrl+B**).
- Use the command `slbuild`, specifying the model reference option, e.g.,:

```
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, this standalone code cannot be used with Model block PIL simulation. If you want to test stand-alone code, you should use the PIL block with an Embedded IDE Link product.

See also the table under “Comparison of the Model Block in PIL Mode and the PIL Block” on page 4-15.

Code Interface for the PIL Block

With an Embedded IDE Link product, you can generate standalone code via a top-level or right-click subsystem build, and automatically create a PIL block to test the code.

Comparison of the Model Block in PIL Mode and the PIL Block

The following table compares the code interfaces and API support for the Model block with PIL and the PIL block (for Embedded IDE Link products).

PIL Feature	Standalone Code Interface (Top-Level or Subsystem Build)	Model Reference RTW Target Code Interface	Target Connectivity API Support
Model block PIL mode	No	Yes	Yes
PIL block	Yes	No (but you can include Model blocks inside wrapper subsystem or model)	No

PIL Modeling Scenarios with the Model Block

You can use the Model block PIL mode to test single components or a whole hierarchy of model reference components. For example, you can select a single leaf component for PIL verification. Later in the development cycle, as your components become integrated into a larger system, you can select a hierarchy of components for PIL verification.

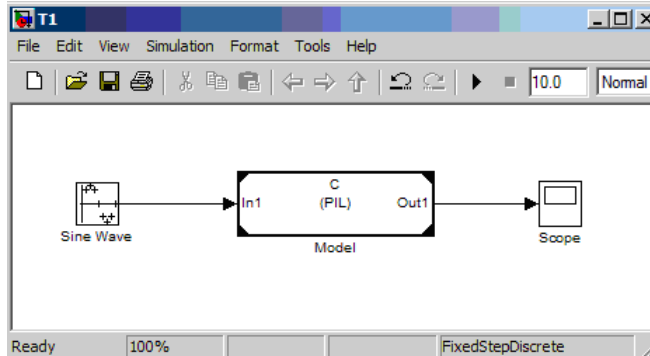
See “Code Interfaces For PIL” on page 4-13 for more information about the standalone and model reference target code interfaces that are referenced in the discussion following.

Your Model block component code must be deployed as part of a standalone executable. The following examples show ways of testing your component.

- “Testing a Model Reference Component in PIL Mode” on page 4-16
- “Deploying via an Atomic Subsystem” on page 4-17
- “Deploying via a Top-Level Model” on page 4-18

Testing a Model Reference Component in PIL 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:

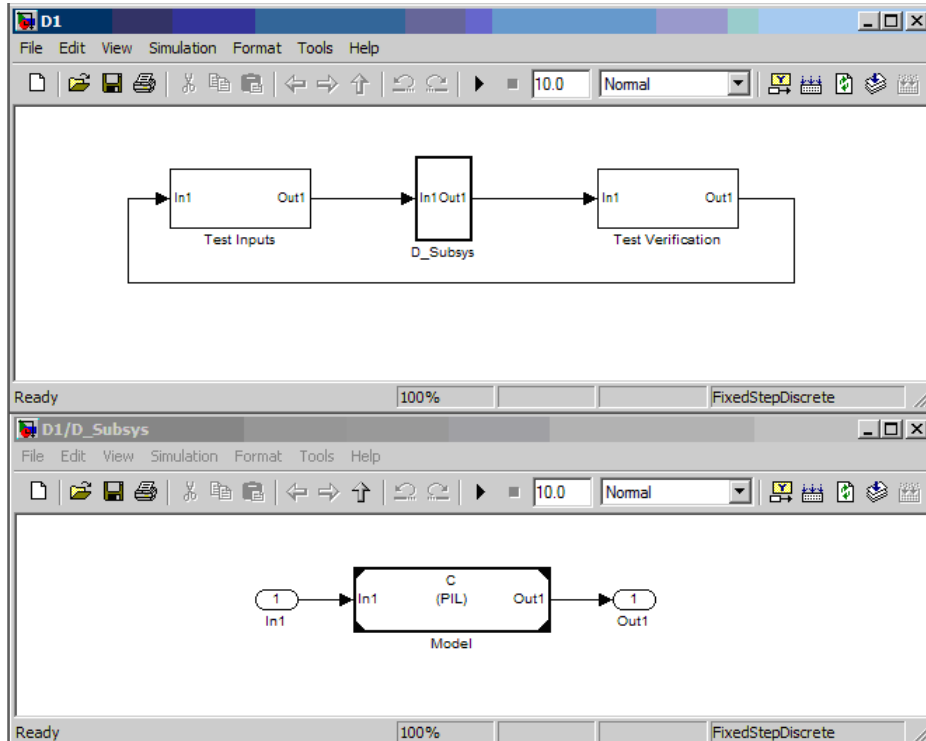
- 1 Set the simulation mode of component C to PIL mode.
- 2 Simulate the model to run component C in PIL mode, and test its RTW model reference target.

Note Simulating the model generates the model reference target code interface for component C, if it does not already exist.

The deployment scenarios in the next sections reuse the model reference target of component C. This reuse ensures that you test exactly the same object code that you will deploy.

Deploying via an Atomic Subsystem

To generate code with the standalone interface for deployment, you can place a Model block inside an atomic subsystem, as shown by model D1.



To create stand-alone code, perform a subsystem build of `D_Subsys`. The stand-alone code calls the RTW model reference target of component `C`.

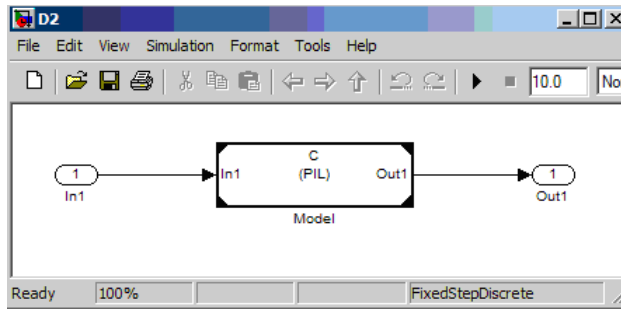
To test the component:

- 1 Set the simulation mode of component `C` to PIL mode.
- 2 Simulate the model to run component `C` in PIL mode, and test its RTW model reference target.

You can place multiple Model blocks and other blocks into the model to deploy a whole system of components.

Deploying via a Top-Level Model

To generate code with the standalone interface for deployment, you can place the Model block inside an top-level model as shown by model D2.

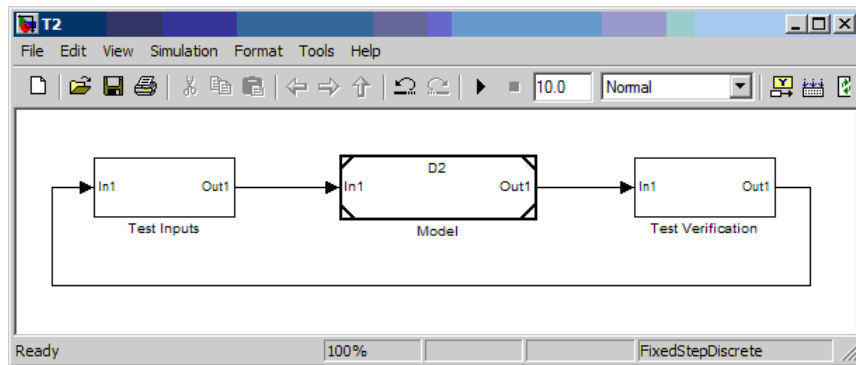


To create stand-alone code, perform a top-level build of D2. The stand-alone code calls the RTW model reference target of component C.

You can place multiple Model blocks and other blocks into the model to deploy a whole system of components.

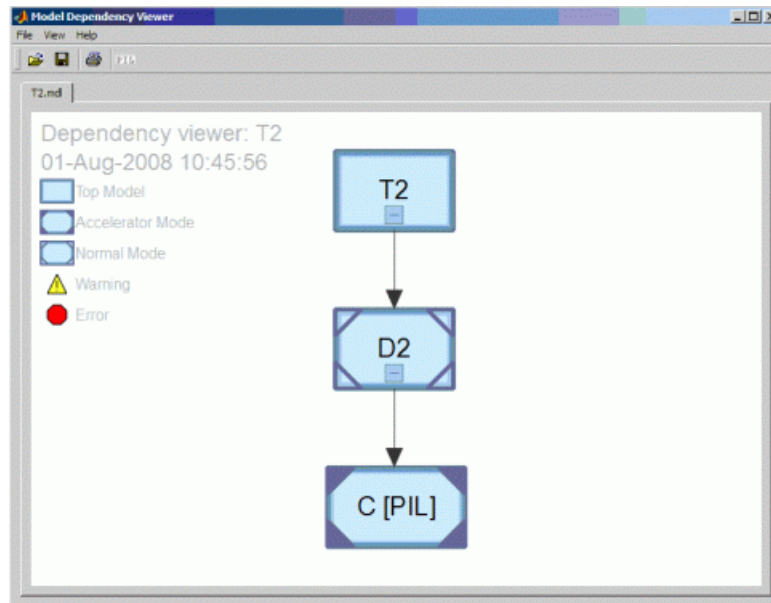
To pass test inputs to component C (running in PIL mode):

- 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 PIL mode and test its RTW model reference target.

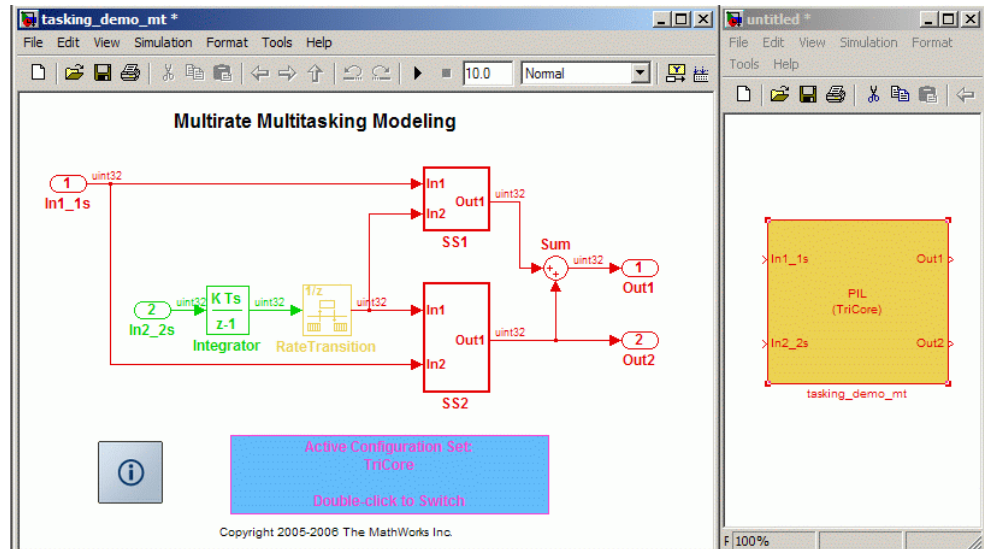
The Model Dependency Viewer shows the model reference hierarchy of T2 and the simulation modes of each Model block component.



Modeling Scenarios with the PIL Block

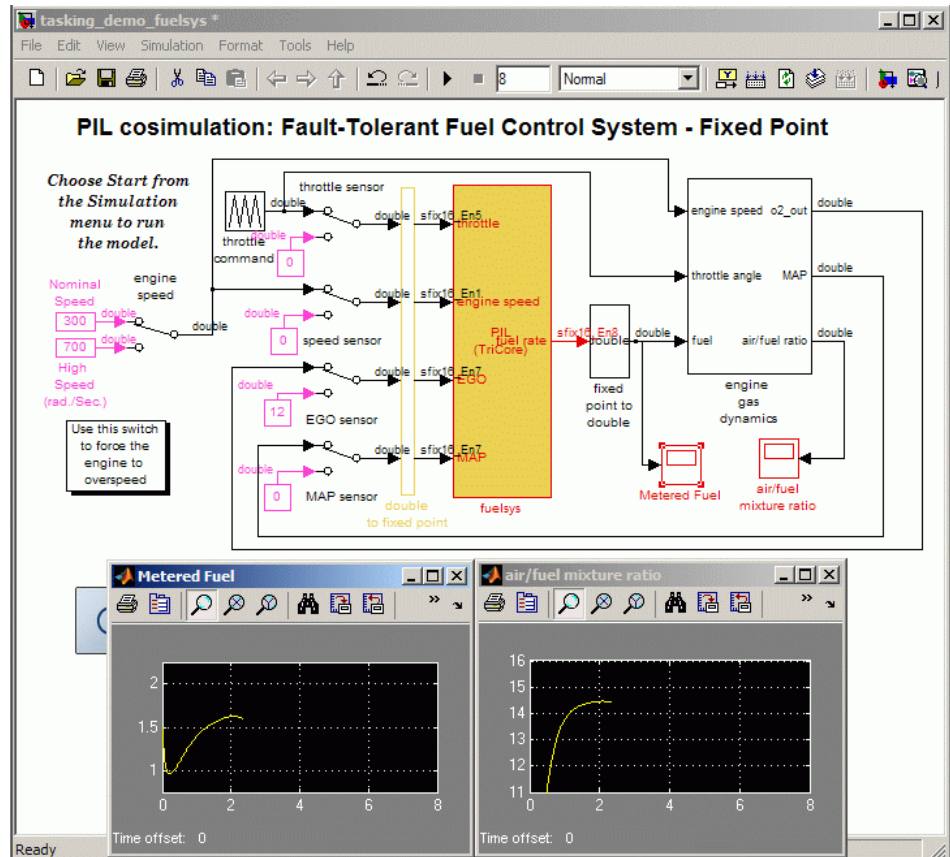
For a top-level build, you get a PIL block that represents the whole model. You can test the deployable standalone object code by calling the PIL block from a test harness model.

The following example shows the PIL block generated for a top-level 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.



Configuring PIL Mode in the Model Block

In this section...
“Using PIL Mode in the Model Block” on page 4-22
“Using the Default PIL Connectivity Configuration” on page 4-22
“PIL Connectivity Configurations ” on page 4-24

Using 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 via PIL connectivity configurations. See “Using the Default PIL Connectivity Configuration” on page 4-22, and “PIL Connectivity Configurations ” on page 4-24.

Using the Default PIL Connectivity Configuration

Real-Time Workshop Embedded Coder software provides a default host-based connectivity configuration that you can use with Model block PIL mode. The default configuration allows you to quickly get started with PIL and explore its capabilities without the need for any third-party tools.

Note You can use the default host-based PIL connectivity configuration with *no configuration required* for any model that meets the requirements in “Compatible Models” on page 4-23..

The host-based connectivity configuration provides a connectivity implementation that:

- 1 Builds a host-based PIL application by invoking a host compiler.
- 2 Launches the PIL application as a separate process on the host machine.

3 Communicates with the PIL application via TCP/IP communications.

See the following sections for more information about host-based PIL:

- “Comparison of Host-Based PIL and SIL” on page 4-23
- “Compatible Models” on page 4-23
- “Host-Based PIL Demo” on page 4-24

For more information about connectivity configurations, see “PIL Connectivity Configurations ” on page 4-24.

Comparison of Host-Based PIL and SIL

The SIL S-function links directly against the generated code, so the generated code runs inside the MATLAB process.

The PIL S-function uses the PIL Connectivity API to communicate over TCP/IP with a PIL application containing the generated code. The PIL application runs as a process on the host machine. The process runs independently of the MATLAB process.

See also the more general comparison in “Comparison of PIL and SIL” on page 4-7.

Compatible Models

You can use the default host-based connectivity configuration (no configuration required) with any Simulink model that meets the following specification:

- 1** The model specifies either the `ert.tlc` or `autosar.tlc` system target file
- 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 Watcom compiler template makefile (`ert_watc.tmf`) is not supported.

- 3 The model specifies one of the following hardware implementation device types:
 - a Generic: 32-bit x86 compatible
 - b Generic: Custom
 - c Generic: MATLAB Host Computer (available only as an Emulation hardware choice)

Note To avoid problems with the host-based configuration, you must close and reopen models containing PIL blocks if you change the template makefile or MEX compiler.

Host-Based PIL Demo

The `rtwdemo_pil` model shows you how to compare results from Normal mode simulation and PIL mode execution for the same referenced model. You simply run the simulation to compare the simulation behavior with the behavior of the corresponding generated code. This uses the default host-based PIL configuration, so the generated code is compiled for and executed on your host workstation.

The model demonstrates PIL where the target is your host machine. To target a different processor, you must create your own PIL configuration. See “PIL Connectivity Configurations” on page 4-24.

PIL Connectivity Configurations

You can use PIL connectivity configurations and the target connectivity API to customize PIL to work with any target environment.

You 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 (i.e., 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. 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 cosimulation, PIL must first determine which of the available connectivity configurations to use. PIL looks for a connectivity configuration that is compatible with the model under test. If multiple or no compatible connectivity configurations are found, then you see an error that reports how to resolve the problem.

You can use these different connectivity configuration options:

- Use the default host-based connectivity configuration.

The target environment is the host machine. This is described in “Using the Default PIL Connectivity Configuration” on page 4-22

- Create a custom connectivity configuration.

The target environment is a different processor.

See “Creating a Connectivity Configuration for Your Target” on page 4-26.

Creating a Connectivity Configuration for Your Target

In this section...
“Overview of the Target Connectivity API” on page 4-26
“Create Your Connectivity API Implementation” on page 4-29
“Register Your Connectivity API Implementation” on page 4-29
“Demos of the Target Connectivity API” on page 4-29

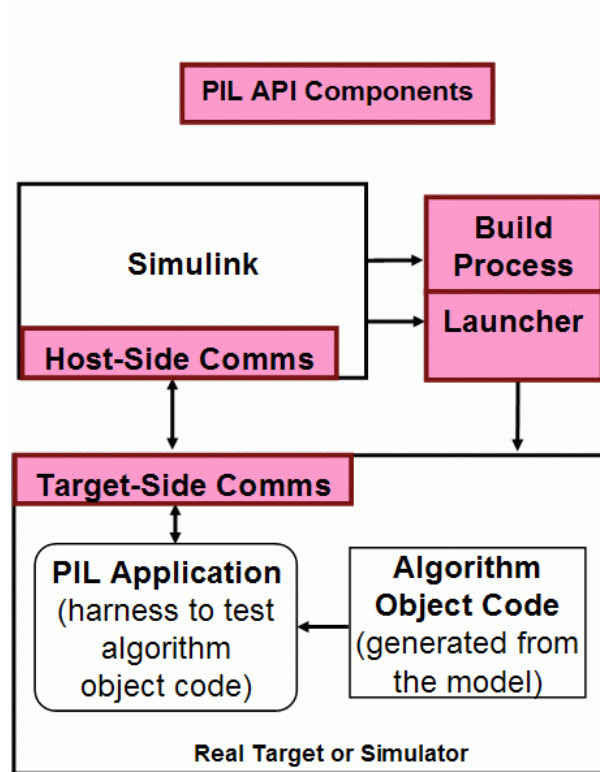
Overview of the Target Connectivity API

- “High-Level Overview” on page 4-26
- “Communications rtiostream API” on page 4-27

High-Level Overview

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 separate driver code running on each of 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 used to send 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. To use the TCP/IP `rtiostream` communications channel, you must provide (or obtain from a third party) target-specific TCP/IP device drivers. For non-TCP/IP communication channels, there is no default implementation provided by Real-Time Workshop software. Therefore 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 M-code.

To see how the `rtiostream` functions fit into the workflow of creating a connectivity implementation, see the next section, “Create Your Connectivity API Implementation” on page 4-29.

Create Your Connectivity API Implementation

To create a target connectivity API implementation, you must create a subclass of `rtw.connectivity.Config`. This class comprises the following other classes.

- 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.
- To configure your `rtiostream` communications implementation:
 - On the target-side, you 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, the driver code must be compiled into a shared library. You load and initialize this shared library by instantiating (or optionally, customizing) `rtw.connectivity.RtIOStreamHostCommunicator`.

See the following reference section for all the classes, methods, and functions in the Target Connectivity API: “Processor-in-the-Loop” in the Real-Time Workshop Embedded Coder Function Reference.

Register Your 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. This step also defines 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

See the following demos for step-by-step examples:

- `rtwdemo_custom_pil`

This M-file 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 M-file 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. This is required for PIL simulation, which requires exchange of data between the Simulink software (running on your host machine) and deployed code (executing on target hardware).

The `rtIOStream` interface provides a generic communication channel that can be implemented in the form of target connectivity drivers for a range of connection types. The demo explains 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 explains how to implement custom target connectivity drivers, e.g., using serial, CAN, or USB for both host and target sides of the communication channel.

PIL Feature Support and Limitations

In this section...
“PIL Functionality Support” on page 4-32
“Code Source Support” on page 4-33
“Block Support” on page 4-34
“Configuration Parameters Support” on page 4-36
“Signal Data Type and I/O Support ” on page 4-40
“Hardware Implementation Support ” on page 4-42
“Other Feature Support” on page 4-47

PIL Functionality Support

Model block processor-in-the-loop (PIL) simulation mode is a Real-Time Workshop Embedded Coder feature.

The PIL block works with the following products:

- Embedded IDE Link™ TS
- Embedded IDE Link MU
- Embedded IDE Link VS
- Embedded IDE Link CC

For more details on the differences between Model block PIL and the PIL block, see “Types of PIL Functionality” on page 4-10.

The following tables describe feature support for Model block PIL and the PIL block. Entries of “Yes” indicate a supported feature.

This is not an exhaustive list of supported features, but provides information on selected features of interest for PIL, especially unsupported features and limitations.

Feature Support	Model Block PIL Mode	PIL Block
Testing of deployment object code	Yes	Yes
Target Connectivity API	Yes	No

Code Source Support

Code Source	Code Interface	Model Block PIL Mode	PIL Block
Top-level model	Standalone	No	Yes
Atomic subsystem	Standalone	No	Yes
Virtual subsystem	Standalone	No	Yes, but recommend atomic subsystem. See “Algebraic Loop Issues” on page 4-37.
Model block	Model reference RTW target	Yes	No, but you can include Model blocks inside wrapper subsystem or model.
Enabled/ Triggered subsystem	Standalone	No	Yes
Export Functions subsystem	Export Functions	No	No
Legacy code	Custom	See “Custom Code Interfaces” on page 4-34.	See “Custom Code Interfaces” on page 4-34.
Embedded MATLAB Coder	Embedded MATLAB Coder	See “Custom Code Interfaces” on page 4-34.	See “Custom Code Interfaces” on page 4-34.

For more information on code interfaces, see “Code Interfaces For PIL” on page 4-13.

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 (e.g., 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 typically flags exceptional conditions during execution of the generated code, such as task overruns. However, task overruns cannot occur during non-real-time PIL cosimulation, so PIL does not need to check for overruns.

The Real-Time Workshop error status can also be set by blocks in the model (such as custom blocks developed by a user). It is a limitation that PIL cannot check this error status and report back errors.

Block Support

Blocks	Model Block PIL Mode	PIL Block
Model block	Yes	No, but you can include Model blocks inside wrapper subsystem or model.
Signal Processing Blockset	Yes	Yes
Video and Image Processing Blockset™	Yes	Yes
Embedded MATLAB block	Yes	Yes
Driver blocks	Yes, but not recommended.	Yes, but not recommended.
To File blocks	No	No
Merge blocks	Cannot connect PIL outputs to Merge blocks. See “Merge Block Issue” on page 4-35.	Cannot connect PIL outputs to Merge blocks. See “Merge Block Issue” on page 4-35.

Run-Time Display Limitation

To Workspace blocks, Scope blocks, and all types of run-time display, such as the display of port values and signal values, have no effect when specified in models executing in PIL mode. The result during simulation is the same as if the constructs did not exist.

Merge Block Issue

If you connect PIL outputs to a Merge block, you will see an error because S-function memory is not reusable. You see an error like the following:

```
The signal from 'mpil_enabled/Subsystem' output port 1 is
required to be persistent, hence this signal cannot be
connected to a Merge block.
```

Configuration Parameters Support

Configuration Parameters	Model Block PIL Mode	PIL Block
ERT-based system target file	Yes	Yes
AUTOSAR system target file	Yes, but see “AUTOSAR Support” on page 4-37.	No
GRT-based system target file	No	No
Generate GRT Interface	No, and see “Missing Code Interface Description File Errors” on page 4-37.	No, and see “Missing Code Interface Description File Errors” on page 4-37.
Function Prototype Control	Yes	Yes
Reusable code format	N/A	Yes
Target Function Library	Yes	Yes
C++	No, and see “Missing Code Interface Description File Errors” on page 4-37.	No, and see “Missing Code Interface Description File Errors” on page 4-37.
Generate ASAP2 file	Yes	Yes
Generate example main	N/A	N/A
MAT-file logging	No	No
’Simplified’ model initialization	No	Yes
Single output/update	Yes, but see “Algebraic Loop Issues” on page 4-37.	Yes, but see “Algebraic Loop Issues” on page 4-37.
Config set ref	Yes	Depends on Emdeded IDE Link product.

Missing Code Interface Description File Errors

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, 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:

- **Generate GRT Interface**
- **Target Language** option **C++ encapsulated**

Make sure these options are not selected.

AUTOSAR Support

There is no PIL support for testing the top level AUTOSAR generated code that contains the AUTOSAR interface.

PIL does support testing components of AUTOSAR models that are modeled as model reference components. These model reference components are implemented as standard model reference RTW targets and do not contain any special AUTOSAR behavior.

For example, you can simulate a top level AUTOSAR model containing PIL components, or you can create a second top model for testing of individual components.

Algebraic Loop Issues

For background on algebraic loops, see:

- “Algebraic Loops” in the Simulink documentation.
- The Algebraic Loops section in “Interactions of the Simulink and Real-Time Workshop Products to Consider” in the Real-Time Workshop documentation.
- The Introduction section in “Nonvirtual Subsystem Code Generation” in the Real-Time Workshop documentation.

There are three ways that PIL cosimulation can introduce algebraic loops that do not exist for a normal simulation:

- “Algebraic Loops Caused by Code Generation for a Virtual Subsystem” on page 4-38
- “Algebraic Loops Caused by “Single output/update function”” on page 4-38
- “Algebraic Loops Caused by PIL Scheduling Limitations” on page 4-38

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 with the simulation behavior.

You should declare virtual subsystems as atomic subsystems to ensure consistent simulation and execution behavior for your model.

See “Nonvirtual Subsystem Code Generation” in the Real-Time Workshop documentation.

Algebraic Loops Caused by “Single output/update function”. The “single output/update function” Real-Time Workshop optimization can introduce algebraic loops because it introduces direct feedthrough via a combined output and update function.

This option is also incompatible 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 PIL Scheduling Limitations. The S-function scheduling mechanism used to execute the PIL component has the following limitations:

- Direct feedthrough is always set to true.
- Separate output and update functions in the PIL component are always executed from the mdlOutputs S-function callback.

These limitations mean that PIL can introduce algebraic loops that do not exist in normal simulation, and wrong results are possible. In this case, you will see a warning or error about the introduced algebraic loop (unless the algebraic loop setting is set to none, in the Configuration Parameters dialog box, under Diagnostics, Solver pane) and PIL results may differ from simulation results.

A workaround is to break the algebraic loop by inserting a Unit Delay block so that the algebraic loop does not occur, then PIL can be used successfully.

Signal Data Type and I/O Support

Signal Data Type and I/O	Model Block PIL Mode	PIL Block
Virtual buses	Not at PIL component boundary	Not at PIL component boundary
Nonvirtual buses	Not at PIL component boundary	Not at PIL component boundary
MUX/DEMUX	Not at PIL component boundary	Not at PIL component boundary
Vector/2D/ Multidimensional	Yes	Yes
Complex data	Yes	Yes
Fixed-point data	Yes	Yes
Complex fixed-point data	Yes	Yes
Fixed-point data type override	Not at PIL component boundary. See “Fixed-Point Tool Data Type Override” on page 4-42.	Not at PIL component boundary. See “Fixed-Point Tool Data Type Override” on page 4-42.
Goto/From I/O	N/A	No. See “Goto/From I/O” on page 4-42.
Data store I/O	No	No
Non-port-based sample times	Yes	Yes
Continuous sample times	No	Not at PIL component boundary.
Outputs with constant sample time	No	Yes
Non-auto-storage classes for signals or parameters	Yes. See “Imported Signal Definitions” on page 4-41.	Yes. See “Imported Signal Definitions” on page 4-41.
Simulink data objects	Yes	Yes
Simulink numeric type and alias type	Yes	Yes

Signal Data Type and I/O	Model Block PIL Mode	PIL Block
Simulink enumerated data	Yes	Yes
Custom storage classes	Yes, but see “Imported Signal Definitions” on page 4-41, and “Unsupported Storage Class Errors” on page 4-41.	Yes, but see “Imported Signal Definitions” on page 4-41, and “Unsupported Storage Class Errors” on page 4-41.

Imported Signal Definitions

You can use I/O signals that specify storage classes with imported data definitions.

Model block PIL. Model block PIL does not define storage for these signals in the PIL application. You must define the storage via custom code included by the component under test or via the PIL `rtw.pil.RtIOStreamApplicationFramework` API.

Similarly, you must define and initialize parameters with imported storage classes.

Even exported global storage classes such as `ExportedGlobal` are imported with respect to the model reference RTW target interfaced with by Model block PIL. Therefore, you must define even signals and parameters with exported global storage classes.

PIL Block . When using the PIL block with signals with imported storage, the signals are defined automatically in the PIL application. For parameters with imported storage, you must define and initialize the parameters.

Unsupported Storage Class Errors

You may see errors like the following if you are using an unsupported custom storage class.

```
The following inports, outports or parameters have
implementations that are not supported by PIL.
```

This error can be caused by using unsupported custom storage classes such as:

- Grouped custom storage classes (e.g., struct, bitfield).
- GetSet custom storage classes

For a right-click build PIL block only, this error can also be caused by Real-Time Workshop not being able to determine memory/storage for a virtual outport. The signal or signals driving the outport are resolved with a storage class. However, the error occurs because the signal's data-type and/or dimensions have been altered. This could be because the signal(s) have been routed via a Mux block or a Data Type conversion block.

Fixed-Point Tool Data Type Override

Signals with data types overridden by the Fixed-Point Tool **Data type override** parameter are not supported at the 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.

Goto/From I/O

For PIL blocks, avoid using Goto blocks for I/O data. For virtual (non-atomic) subsystems, the right-click PIL build transforms root-level Goto blocks into outports and From blocks into inports. The resulting PIL block has extra I/O ports and you must rework the model to connect it. To avoid this situation, do not use Goto blocks.

Hardware Implementation Support

Hardware Implementation	Model Block PIL Mode	PIL Block
Different host/target datatype size	Not at PIL component boundary. See “Hardware Implementation Settings” on page 4-43.	Not at PIL component boundary. See “Hardware Implementation Settings” on page 4-43.

Hardware Implementation	Model Block PIL Mode	PIL Block
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 the Hardware Implementation settings (in the Simulink Configuration Parameters dialog box) are configured correctly for the target environment.

Byte ordering must be specified for non-8-bit targets.

See the following sections for more information:

- “Host/Target Data Type Size Mismatch” on page 4-43
- “Model Block PIL Data Type Size Mismatch Issues” on page 4-44
- “PIL Block Data Type Size Mismatch Issues” on page 4-45

Host/Target Data Type Size Mismatch. Only data types that have the same size on the host and the target are supported at the PIL I/O boundary.

The data types used at the PIL I/O boundary are restricted based on the following rule : the data type is supported for PIL 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:

- Single/double on targets with 24-bit floating point types
- Double on targets with 32-bit double, i.e., the same size as single

Warning It is not always possible for PIL to detect unsupported data types (see Model Block PIL and PIL Block sections below for details). In such cases, data transfer between host and target is incorrect and unexpected data transfer errors occur during cosimulation.

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 also has the benefit of being more efficient.

Model Block PIL Data Type Size Mismatch Issues. PIL mode on the Model Block 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 Model block PIL does not detect violations of these assumptions. If the settings for the target environment violate any of these assumptions, then unexpected data transfer errors occur during cosimulation.

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 also has the benefit of being more efficient.

Some known violations with predefined hardware implementation settings:

- Unsupported word addressable targets: TI C2000, Freescale DSP563xx

- Unsupported datatypes owing to word order: TASKING Infineon C166 (single and double have flipped word order)
- Unsupported datatypes owing to size mismatch: TASKING 8051 (double > is only 4 bytes)

PIL Block Data Type Size Mismatch Issues. Embedded IDE Link products register 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 the PIL block to support target environments with unusual hardware characteristics.

The PIL block 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 cosimulation.

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 also has the benefit of being more efficient.

Some target compilers allow the sizes of target data types to be changed from their default size. For example, an IEEE double datatype 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 Embedded IDE Link products is typically statically defined to match the compiler's default data type size, and therefore does not support the ability to change the data type size from that default size.

Warning If a nondefault compiler configuration is used, it is possible that the target data type sizes registered by Embedded IDE Link products 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 cosimulation.

To resolve this issue, either:

- Do not use compiler options that change the default size of target data types.
- If your aim is to treat double-precision floating-point types (8 bytes) as single-precision floating-point types (4 bytes), then use the single data type in Simulink rather than double.

Other Feature Support

Other Features	Model Block PIL Mode	PIL Block
Multiplatform support (e.g. Linux)	Yes	Depends on Embedded IDE Link product.
Execution profiling	Depends on PIL configuration. See “Profiling Support” on page 4-47.	Yes (if Embedded IDE Link support)
Stack profiling	Depends on PIL configuration. See “Profiling Support” on page 4-47.	Yes (if Embedded IDE Link support)
C code coverage report	Depends on PIL configuration. See “Profiling Support” on page 4-47.	Yes (if Embedded IDE Link support)
Tunable parameters	No	No
Logging of internal data	No	No

Profiling Support

PIL does not provide integrated support for execution profiling, stack profiling, or code coverage reporting. However, you can set up your PIL configuration’s target connectivity API implementation to retrieve the information from a third-party tool, if available for your target.

Developing Models and Code That Comply with Industry Standards and Guidelines

- “What Are the Standards and Guidelines?” on page 5-2
- “Developing Models and Code That Comply with MAAB Guidelines” on page 5-4
- “Developing Models and Code That Comply with MISRA C Guidelines” on page 5-5
- “Generating Code That Complies with AUTOSAR Standards” on page 5-6
- “Developing Models and Code That Comply with the IEC 61508 Standard” on page 5-25
- “Developing Models and Code That Comply with the DO-178B Standard” on page 5-28

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 5-4
Guidelines: Use of the C Language in Critical Systems (MISRA C ^{®4})	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 5-5
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 • “Generating Code That Complies with AUTOSAR Standards” on page 5-6

4. 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 5-25
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 5-28

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 by running the Simulink Model Advisor on MathWorks Automotive Advisory Board checks.

- 1 Open your model or subsystem.
- 2 Start the Model Advisor.
- 3 In the **Task Hierarchy**, expand **By Product > Simulink Verification and Validation > Modeling Standards > MathWorks Automotive Advisory Board Checks**.
- 4 Select the checks that you want to enable. To generate an HTML report that shows the check results, also select **Show report after run**.
- 5 Click **Run Selected Checks**. The Model Advisor processes the checks and displays the results.
- 6 In the Model Advisor window, review the check results and make any necessary changes. To see detailed results for a specific check, select the check in the **Task Hierarchy**. The results appear in the right pane.

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 (MISRA⁵) 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.

For information about using Real-Time Workshop Embedded Coder software within MISRA C guidelines, see Technical Solution 1-1IFP0W on the MathWorks Web site.

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

Generating Code That Complies with AUTOSAR Standards

In this section...

“Overview of AUTOSAR Support” on page 5-6

“Importing an AUTOSAR Software Component” on page 5-8

“Using the Configure AUTOSAR Interface Dialog Box” on page 5-10

“Configuring Ports for Basic Software and Error Status Receivers” on page 5-14

“Configuring Calibration Parameters” on page 5-15

“Modifying and Validating an Existing AUTOSAR Interface” on page 5-16

“Exporting a Single Runnable Entity AUTOSAR Software Component” on page 5-17

“Configuring Multiple Runnable Entities” on page 5-19

“Configuring AUTOSAR Options Programmatically” on page 5-21

“Verifying the AUTOSAR Code Using Software-in-the-Loop Testing” on page 5-21

“Limitations and Tips” on page 5-22

“Migrating AUTOSAR Development Kit Models” on page 5-23

“Demos and Further Reading” on page 5-23

Overview of AUTOSAR Support

Real-Time Workshop Embedded Coder software supports *AUTomotive Open System ARchitecture*, AUTOSAR, an open and standardized automotive software architecture. AUTOSAR is jointly developed by automobile manufacturers, suppliers and tool developers.

You can use Real-Time Workshop Embedded Coder software to import AUTOSAR software components into Simulink models, generate AUTOSAR-compliant code, and export AUTOSAR-compliant XML files. You can configure your models by GUI, command-line functions, or AUTOSAR-compliant XML files.

The following table maps different aspects of working with AUTOSAR components to the following sections.

Workflow	Description and Links to Sections
Importing an AUTOSAR software component.	You can import AUTOSAR software component description XML files, including calibration parameters for use in your model. “Importing an AUTOSAR Software Component” on page 5-8 “Configuring Calibration Parameters” on page 5-15
Configuring for AUTOSAR-compliant code generation.	To configure AUTOSAR code generation options and XML export options, you can use the Configure AUTOSAR Interface dialog box, or configure programmatically. “Using the Configure AUTOSAR Interface Dialog Box” on page 5-10 “Configuring Ports for Basic Software and Error Status Receivers” on page 5-14 “Configuring AUTOSAR Options Programmatically” on page 5-21
Validating a modified AUTOSAR interface.	You can validate your interface from the Configure AUTOSAR Interface dialog box or programmatically. “Using the Configure AUTOSAR Interface Dialog Box” on page 5-10 “Modifying and Validating an Existing AUTOSAR Interface” on page 5-16

Workflow	Description and Links to Sections
Exporting an AUTOSAR Software Component.	After configuring your AUTOSAR export options, generate code as usual to export your AUTOSAR component. Building the subsystem or model generates the code and XML according to your customizations. “Exporting a Single Runnable Entity AUTOSAR Software Component” on page 5-17 “Configuring Multiple Runnable Entities” on page 5-19
Verification.	“Verifying the AUTOSAR Code Using Software-in-the-Loop Testing” on page 5-21

Importing an AUTOSAR Software Component

Use the `arxml.importer` class to parse an AUTOSAR software component description file (for example, exported from a tool such as DaVinci System Architect from Vector Informatik GmbH) and import into a Simulink model for configuration, code generation, and export to XML. See “AUTOSAR” in the Real-Time Workshop Embedded Coder Function Reference documentation for a full list of methods. Typical uses of functions follow these steps:

- 1 Call `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 at the command line 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'
```

You can 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, you must specify the file that contains the software component of interest.

- 2 Use the `setDependencies` method if you need to specify additional dependent XML files containing the information that completes the software component description (e.g., data types, interfaces). You can specify a cell array of files or a single file.

You need to complete specifying dependencies only for components that you intend to import into Simulink.

- 3 Call one of the following methods to import a parsed atomic software component into a Simulink model. If you have not specified all dependencies for the components, you will see errors.

- `createComponentAsSubsystem` — Creates and configures a Simulink subsystem skeleton corresponding to the specified atomic software component description.
- `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')
```

- `createCalibrationComponentObjects` — Creates Simulink calibration objects corresponding to the specified AUTOSAR calibration component description.

For example:

```
[success] = createCalibrationComponentObjects(importerObj,  
CreateSimulinkObject, 'true')
```

See also the limitation “Cannot Import Internal Runnable Behavior” on page 5-22.

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 5-10, or configure programmatically (see “Configuring AUTOSAR Options Programmatically” on page 5-21).

See the demo Import and Export an AUTOSAR Software Component, which demonstrates how to import, modify, and export AUTOSAR software components.

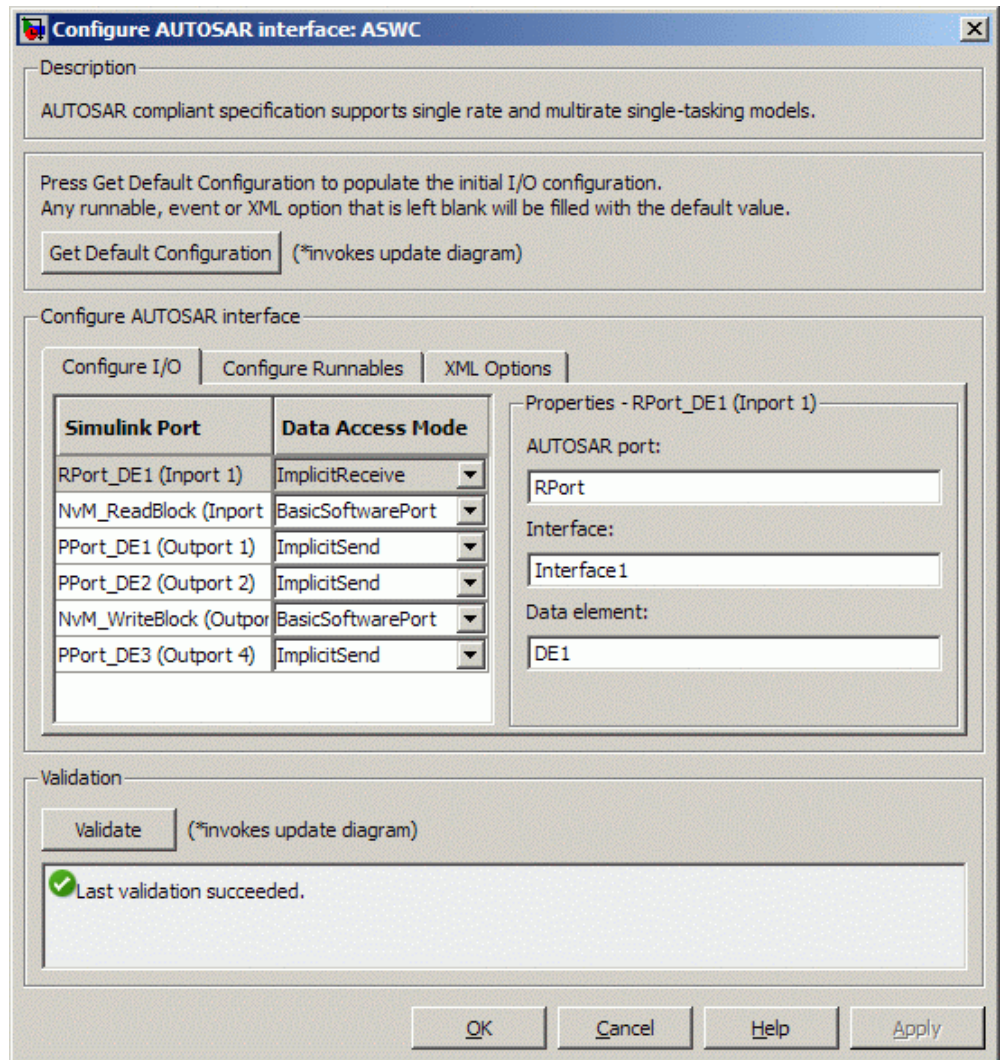
Using the Configure AUTOSAR Interface Dialog Box

You can 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 programmatically control all AUTOSAR options.

You can open the Configure AUTOSAR Interface dialog box (in any model using the `autosar.tlc` system target file) 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.

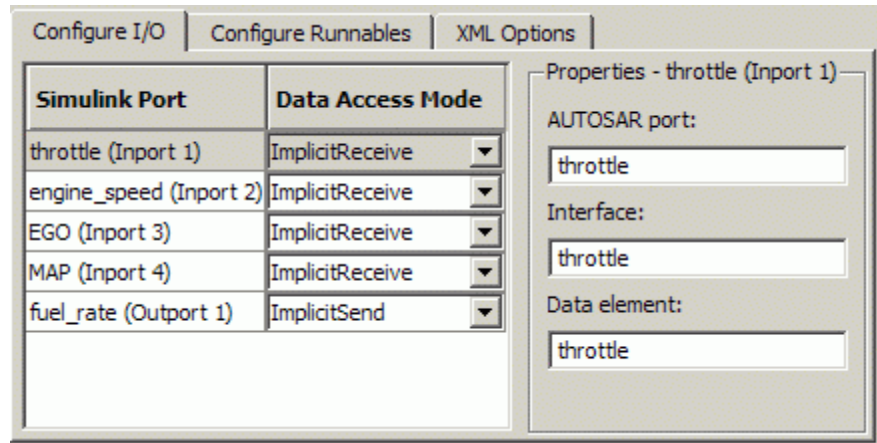


To configure your AUTOSAR options:

- 1 Click **Get Default Configuration** to populate the controls for your model.

The runnable entity 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.

- 2 Use the controls in the **Configure AUTOSAR Interface** pane to make changes to your AUTOSAR code generation options and XML export options, e.g., send/receive communication options such as port and interface names, data access modes, runnable entity, initialization, and periodic function names.
 - On the **Configure I/O** tab, you can 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 either **Implicit**, where data is buffered by the run-time environment (RTE), or **Explicit** where data is not buffered and hence not deterministic.

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 alone. 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_'.

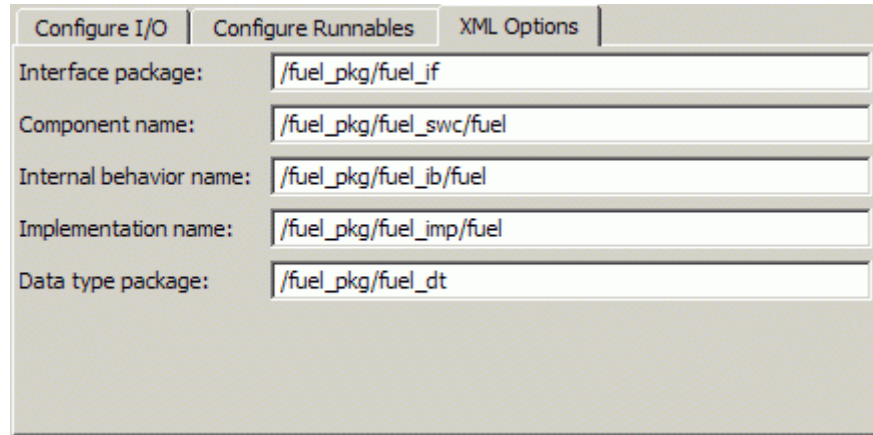
For more information on all these options, see “AUTOSAR” in the Real-Time Workshop Embedded Coder Function Reference documentation.

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 5-14.

- On the **Configure Runnables** tab, you can specify the names of your runnable entity, 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, you can specify the names and package paths of the XML files you will publish when you generate code. For more details about these files, see “Exporting a Single Runnable Entity AUTOSAR Software Component” on page 5-17.



The image shows a configuration dialog box with four tabs: "Configure I/O", "Configure Runnables", "XML Options", and an unlabeled tab. The "Configure Runnables" tab is selected. Below the tabs are five text input fields, each with a label on the left and a value in the box:

Label	Value
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

- 3 After you have configured your options, click **Validate**. This step calls `runValidation`. If there are problems, you see messages explaining why the configuration is invalid.

Note See `runValidation (AUTOSAR)` in the Real-Time Workshop Embedded Coder Function Reference documentation for information on all validation checks.

- 4 If validation succeeds, click **OK** to return to the Configuration Parameters dialog box.
- 5 Save your model and then generate code as usual to export your AUTOSAR component.

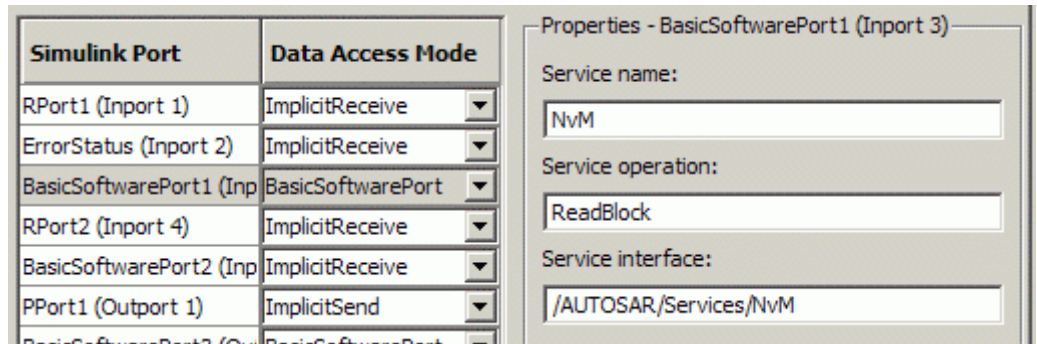
Configuring Ports for Basic Software and Error Status Receivers

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, you can specify the **Data Access Mode** of every port.

- You can designate inports and outports as access points to basic software.

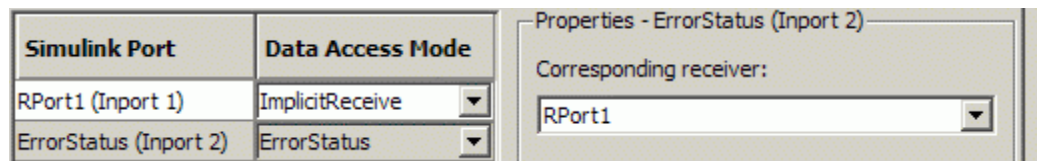
If you select **Basic Software**, you must specify the service name, operation and interface. The service name and operation must be valid AUTOSAR identifiers, and the service interface must be a valid path of the form `AUTOSAR/Service/servicename`.



After you export your AUTOSAR components, remember that you need to include your service interface definition XML file to import correctly into an authoring tool.

- You can 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 Calibration Parameters

To import calibration parameters, use the importer method `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, you must 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, remember you need to include your calibration interface definition XML file to import correctly into an authoring tool.

Modifying and Validating an Existing AUTOSAR Interface

To validate your AUTOSAR interface,

- 1 Get the handle to an existing model-specific `RTW.AutosarInterface` object that is attached to your loaded Simulink model by entering:

```
obj = RTW.getFunctionSpecification(modelName)
```

Where *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 by entering:

```
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 `runValidation (AUTOSAR)`.

Note See `runValidation` (AUTOSAR) in the Real-Time Workshop Embedded Coder Function Reference documentation for information on all validation checks.

- 4 If validation succeeds, save your model and then generate code.

Exporting a Single Runnable Entity AUTOSAR Software Component

After configuring your AUTOSAR export options programmatically or in the GUI, generate code as usual to export your AUTOSAR component.

Building the subsystem or model generates the code and XML according to your customizations.

The software component C code and the following XML files are exported to the build directory:

Filename	Description
<code>modelName_behavior.arxml</code>	Specifies the software component internal behavior.
<code>modelName_implementation.arxml</code>	Specifies the software component implementation.
<code>modelName_interface.arxml</code>	Specifies the software component interfaces, including extra interfaces.
<code>modelName_component.arxml</code>	Specifies the software component type, including additional ports added to the Simulink model.
<code>modelName_datatype.arxml</code>	Specifies the software component data types, including any modified or additional data types.

You can then merge this 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.

To use the DaVinci AUTOSAR authoring tool, call this function *before* generating code:

```
autosar_davinci_configure.m
```

This function configures your model to use the `autosar.tlc` system target file and AUTOSAR schema 2.1, and when you generate code, four XML files are exported. The component type and internal behavior XML files are combined into one file for use in DaVinci, called `modelName_component.arxml`.

See the `rtwdemo_autosar_legacy_script` AUTOSAR Code Generation demo for an example that explains how to generate AUTOSAR-compliant code and export AUTOSAR software component description XML files from a Simulink model.

Selecting an AUTOSAR Schema

The default AUTOSAR schema version is 2.1. 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.

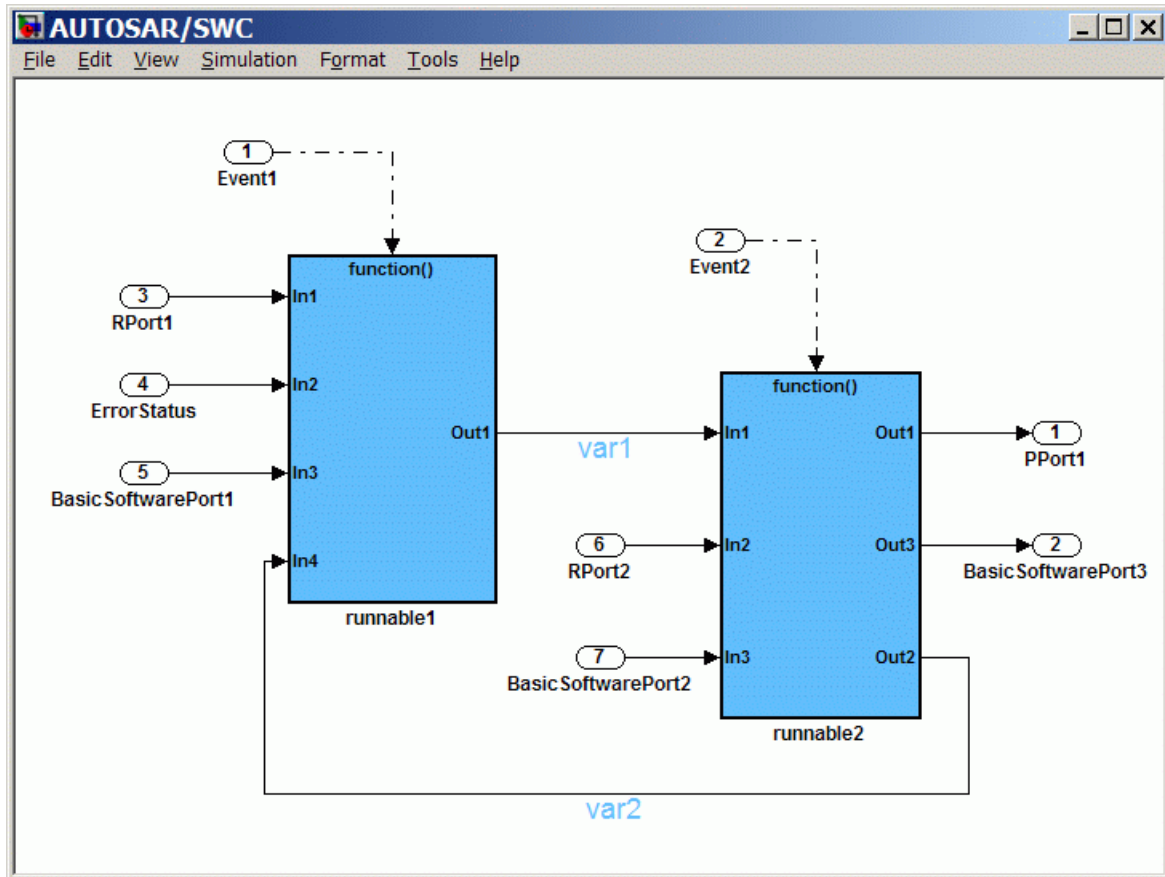
- 2 Select a schema version (2.1 or 2.0) for generating XML files.

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.

Configuring Multiple Runnable Entities

You can use subsystems to explicitly model multiple runnable entities in a single AUTOSAR software component. You can map AUTOSAR runnable entities to function-call subsystems by using the Export Functions feature. If your top-level model contains function-call subsystems like the following example, you can use the Export Functions command to specify that you want each subsystem to represent an AUTOSAR runnable entity.

You must specify an AUTOSAR interface for each function call subsystem being exported as a runnable entity. 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 runnable entities 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 entity for each subsystem. In the example shown, `runnable1` and `runnable2` are both runnable entities. The build also creates an additional runnable entity at code generation time to aggregate the initialization functions for each of the function-call subsystems.

Inter-runnable variables communicate data between runnable entities to ensure data integrity. You define these inter-runnable variables by the signal lines connecting subsystems (var1 and var2 in the example shown). Double-click to edit these signal names as appropriate 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 entity if a block uses time. The timer increments at each execution of the runnable entity. You specify the timer resolution with the **Execution period** field in the AUTOSAR Model Interface dialog box. You specify the **Application life span (days)** in the Configuration Parameters under Optimization. The timer data type is chosen based on your specified application life span and execution period.

See the `rtwdemo_autosar_multirunnables_script` AUTOSAR Code Generation for Multiple Runnable Entities demo, which shows how to configure and generate AUTOSAR-compliant code and export AUTOSAR software component description XML files for a Simulink model with multiple runnables entities.

Configuring AUTOSAR Options Programmatically

To programmatically control AUTOSAR options, 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 Using Software-in-the-Loop Testing

A common technique to verify the generated code is to wrap the generated code in an S-function. This technique allows you to verify the generated code in simulation. The AUTOSAR target automatically configures the generated S-function to route simulation data using AUTOSAR run-time environment (RTE) API calls.

- 1 Configure your model for software-in-the-loop (SIL) testing by setting these two configuration parameters:

```
set_param( modelName, 'GenerateErtSFunction', 'on' );  
set_param( modelName, 'GenCodeOnly', 'off' )
```

- 2 Generate code to build the SIL block.
- 3 Once the SIL block has been built, replace the existing component in your model with the new block.
- 4 Simulate the model and check the output to verify the code produces the same data as the original subsystem.

Limitations and Tips

Cannot Import Internal Runnable Behavior

Input/Output configuration (e.g., implicit or explicit) is imported at component level only. Internal runnable behavior is not parsed. This means any I/O information stored at the runnable level (e.g., 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 the Configure AUTOSAR Interface dialog box.

Cannot Copy Subsystem Blocks Without Losing Interface Information

If you copy a subsystem block and paste it 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. You must configure your model with the **Get Default Configuration** button or the `getDefaultConf` (AUTOSAR) 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.

Cannot Save Importer Objects in MAT-Files

If you try to save an `arxml.importer` object in a MAT-file, then all information is lost. If you reload the MAT-file, then the object is null (`handle = -1`). This is because of the Java™ objects that compose the `arxml.importer` object.

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 and it is advisable to save the migrated model to a different file name.

Demos and Further Reading

AUTOSAR Demos

See the demos in the following table for detailed explanations of AUTOSAR workflows with Real-Time Workshop Embedded Coder software.

Demo	Description
AUTOSAR Code Generation: <code>rtwdemo_autosar_legacy_script</code>	Demonstrates how to generate AUTOSAR-compliant code and export AUTOSAR software component description XML files from a Simulink model

Demo	Description
AUTOSAR Code Generation for Multiple Runnable Entities: rtwdemo_autosar_multirunnables_script	Demonstrates how to configure and generate AUTOSAR-compliant code and export AUTOSAR software component description XML files for a Simulink model with multiple runnables entities.
Import and Export an AUTOSAR Software Component: rtwdemo_autosar_roundtrip_script	Demonstrates 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/>
- *Applying Simulink to AUTOSAR* on the AUTOSAR Web site:
http://www.autosar.org/download/AUTOSAR_SimulinkStyleguide.pdf

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 5-25

“Checking for IEC 61508 Standard Compliance Using the Model Advisor” on page 5-25

“Validating Traceability” on page 5-29

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 on IEC 61508 checks.

- 1 Open your model or subsystem.
- 2 Start the Model Advisor.

- 3** In the **Task Hierarchy**, expand **By Product > Simulink Verification and Validation > Modeling Standards > IEC 61508 Checks** or **By Task > Modeling Standards for IEC 61508**.
- 4** Select the checks that you want to enable.
- 5** Select **Show report after run** if you want to display an HTML report that shows the check results. Alternatively, later you can click the report link in the **Last Report** section of the results pane. In either case, you can save and print the resulting report for review or archiving purposes.
- 6** Click **Run Selected Checks**. The Model Advisor processes the checks and displays the results.
- 7** In the Model Advisor window, review the check results and make any necessary changes. To see detailed results for a specific check, select the check in the **Task Hierarchy**. The results appear in the right pane.

Note If your model uses model referencing, apply Model Advisor checks to all referenced models before applying them to the top-level model.

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.

To...	Use...
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 5-28

“Checking for Standard Compliance Using the Model Advisor” on page 5-28

“Validating Traceability” on page 5-29

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 on DO-178B checks.

- 1 Open your model or subsystem.
- 2 Start the Model Advisor.

- 3** In the **Task Hierarchy**, expand **By Product > Simulink Verification and Validation > Modeling Standards > DO-178B Checks** or **By Task > Modeling Standards for DO-178B**.
- 4** Select the checks that you want to enable.
- 5** Select **Show report after run** if you want to display an HTML report that shows the check results. Alternatively, later you can click the report link in the **Last Report** section of the results pane. In either case, you can save and print the resulting report for review or archiving purposes.
- 6** Click **Run Selected Checks**. The Model Advisor processes the checks and displays the results.
- 7** In the Model Advisor window, review the check results and make any necessary changes. To see detailed results for a specific check, select the check in the **Task Hierarchy**. The results appear in the right pane.

Note If your model uses model referencing, apply Model Advisor checks to all referenced models before applying them to the top-level model.

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.

To...	Use...
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

Custom Storage Classes

- “Introduction to Custom Storage Classes” on page 6-2
- “Custom Storage Class Memory Sections” on page 6-5
- “Predefined Custom Storage Classes” on page 6-6
- “Changing Custom Storage Classes” on page 6-8
- “Setting Custom Storage Class Properties” on page 6-9
- “Generating Code with CSCs” on page 6-11
- “Creating Packages with CSC Definitions” on page 6-16
- “Designing Custom Storage Classes and Memory Sections” on page 6-20
- “Defining Advanced Custom Storage Class Types” on page 6-42
- “GetSet Custom Storage Class for Data Store Memory” on page 6-46
- “Setting Code Generation Options for Custom Storage Classes” on page 6-50
- “Custom Storage Class Limitations” on page 6-51
- “Older Custom Storage Classes (Prior to Release 14)” on page 6-52

Introduction to Custom Storage Classes

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 built-in storage classes for use with all targets. Examples of built-in storage classes are `Auto`, `ExportedGlobal`, and `ImportedExtern`. 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. Built-in storage classes are discussed in detail in the “Working with Data” section of 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. For example, you may need 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.

Real-Time Workshop Embedded Coder *custom storage classes* (CSCs) provide extended control over the constructs required to represent data in an embedded algorithm. CSCs extend the built-in storage classes provided by the Real-Time Workshop software. 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 associate your own CSC definitions with classes contained in the package. For information about the Data Class Designer, see “Subclassing Simulink Data Classes” and “Creating Packages with CSC Definitions” on page 6-16.
- 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 Objects and Code Generation” 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 additional CSCs that are tailored to your code generation requirements. The Custom Storage Class Designer provides a graphical user interface that lets you implement CSCs quickly and easily. You can use your CSCs in code generation immediately, without any Target Language Compiler (TLC) or other programming.

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*.

A CSC registration file is provided 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. If you want to customize or extend the predefined CSCs, or create CSCs for use with data class packages other than the `Simulink` package, you can by using the Custom Storage Class Designer. The Custom Storage Class Designer is described in “Designing Custom Storage Classes and Memory Sections” on page 6-20.

The demo `rtwdemo_importstruct` shows how the Custom Storage Class capabilities work together. Using the techniques shown in the demo, you can specify and change the values of any number of parameters with a single

C statement. The statement assigns to a pointer variable the address of a structure that defines the desired values. Assigning the address of a different structure changes all the values at once.

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 “Editing Memory Section Definitions” on page 6-35 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. More information about using memory sections with custom storage classes, and complete information about using them in other ways, appears in Chapter 7, “Memory Sections”.

Predefined Custom Storage Classes

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

The `RTWInfo` field of the `Simulink.Signal` and `Simulink.Parameter` classes (and of any subclasses derived from these classes) contains two properties that support use of CSCs in code generation:

- **CustomStorageClass:** To assign a custom storage class to a signal or parameter object, you set the `RTWInfo.CustomStorageClass` property to one of the available CSC names and `RTWInfo.StorageClass` to `Custom`. Predefined Simulink® CSCs for Signal and Parameter Objects on page 6-7 lists the predefined set of CSCs provided by the Real-Time Workshop Embedded Coder software.
- **CustomAttributes:** Some CSCs have *instance-specific* properties that define attributes of individual objects (or instances) of that class. The `RTWInfo.CustomAttributes` property lets you define these attributes. For example, you can pack signal objects of class `Struct` into different data structures in the generated code by setting the `RTWInfo.CustomAttributes.StructName` property for each object. Instance-Specific Properties for CSCs on page 6-10 lists instance-specific properties for the predefined set of CSCs provided by the Real-Time Workshop Embedded Coder software.

Some CSCs are valid for parameter objects but not signal objects and vice versa (even though they are not defined in predefined CSCs). For example, you can assign the storage class `Const` to a parameter object. This storage class is not valid for signals, because, in general, signal data is not constant. The next table indicates whether each class is valid for parameter and/or signal objects.

Predefined Simulink CSCs for Signal and Parameter Objects

Class Name (Enumerated)	Available for Signals	Available for Parameters	Purpose
BitField	Y	Y	Generate a struct declaration that embeds Boolean data in named bit fields.
Const	N	Y	Generate a constant declaration with the const type qualifier.
ConstVolatile	N	Y	Generate declaration of volatile constant with the const volatile type qualifier.
Default	Y	Y	Default is a placeholder CSC that the code generator assigns to the RTWInfo.CustomStorageClass property of signal and parameter objects when they are created. You cannot edit the default CSC definition.
Define	N	Y	Generate #define directive.
ExportToFile	Y	Y	Generate header (.h) file, with user-specified name, containing global variable declarations.
ImportFromFile	Y	Y	Generate directives to include predefined header files containing global variable declarations.
Struct	Y	Y	Generate a struct declaration encapsulating parameter or signal object data.
Volatile	Y	Y	Use volatile type qualifier in declaration.

Changing Custom Storage Classes

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 an M-file (`csc_registration.m`). A built-in package defines custom storage classes in both a P-file and a functionally equivalent M-file. A user-defined package initially defines custom storage classes only in an M-file.

P-files take precedence over M-files, so when MATLAB looks for a package's CSC registration file and finds both a P-file and an M-file, MATLAB loads the P-file and ignores the M-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 M-file. M-files are editable, so all CSC Designer editing capabilities are enabled for CSCs stored in an M-file.

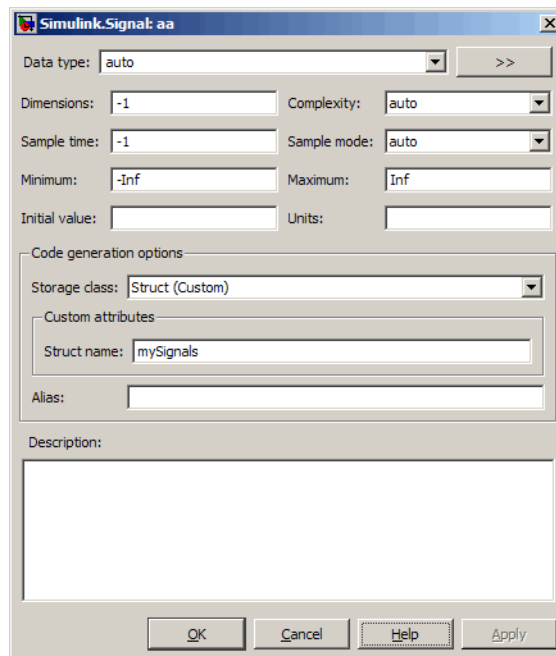
Because CSC definitions for a built-in package exist in both a P-file and an M-file, they are uneditable. You can make the definitions editable by deleting the P-file, but The MathWorks discourages modifying CSC registration files or any other files under `matlabroot`. The preferred technique is to create 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 an M-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 M-file. However, you should preserve the M-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 M-files that define CSCs. However, The MathWorks discourages this practice, which is vulnerable to syntax errors and can give unexpected results if a shadowing P-file exists. When MATLAB finds an older P-file that shadows a newer M-file, it displays a warning in the MATLAB Command Window.

Setting Custom Storage Class Properties

You can set the `CustomStorageClass` and `CustomAttributes` properties (if applicable) of signal and parameter objects by using the data object dialog box. This dialog box appears in the right pane of the Model Explorer. Alternatively, you can launch the dialog box independently by right-clicking the relevant object in the center pane of the Model Explorer and choosing **Properties**. The following figure shows a Model Explorer properties view of a signal object, `aa`. The **Storage class** menu sets the `RTWInfo.CustomStorageClass` property for the object. In this case the **Storage class** field specifies the custom storage class `Struct`. The `Struct` storage class has the instance-specific property **Struct name** (`RTWInfo.CustomAttributes.StructName`). This property is set to `mySignals`.



You can also set these properties with MATLAB commands, for example:

```
aa = Simulink.Signal;  
aa.RTWInfo.StorageClass = 'Custom';
```

```
aa.RTWInfo.CustomStorageClass = 'Struct';
aa.RTWInfo.CustomAttributes.StructName = 'mySignals';
```

When setting CSC-related RTWInfo properties with MATLAB commands, make sure that the RTWInfo.StorageClass property is set to `Custom`. If you set this property to another value, the custom storage properties are ignored. If you set RTWInfo.customStorageClass without first setting RTWInfo.StorageClass to `Custom`, the code generator displays a warning at the MATLAB command line. If you configure these properties with the Simulink Model Explorer, RTWInfo.StorageClass is automatically set to the correct value.

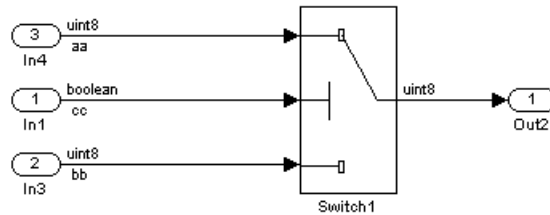
In the generated code, storage for the signal `aa` is allocated within a `struct` named `mySignals`. This is demonstrated in “Generating Code with CSCs” on page 6-11.

Instance-Specific Properties for CSCs

Class Name (Enumerated)	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.
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.

Generating Code with CSCs

This section presents a simple example of code generation with CSCs, based on the model shown in this figure.



This example uses signal objects, but the procedure for generating code from parameter objects (or from any class of objects that supports CSCs) is almost the same. (If you plan to use CSCs with parameter objects, see “Setting Code Generation Options for Custom Storage Classes” on page 6-50 for the correct use of the **Inline parameters** option.)

The model contains three named signals (aa, bb, and cc). Using the predefined **Struct** CSC, this example packs these signals into a named struct, `mySignals`, in the generated code. The struct declaration is then exported to externally written code.

To generate the struct, you must instantiate `Simulink.Signal` objects that are associated (by name) with the signals in the model, and assign the appropriate storage class to the `Simulink.Signal` objects. In this case, the code generator uses the **Struct** custom storage class. After these objects are configured, code generation can proceed.

Set Model Properties

Before configuring the signal objects, make sure you deselect the **Ignore custom storage classes** option in the **Real-Time Workshop** pane of the active configuration set.

Instantiate Signal Objects

The next step is to instantiate signal objects. You can do this with MATLAB commands as shown below.

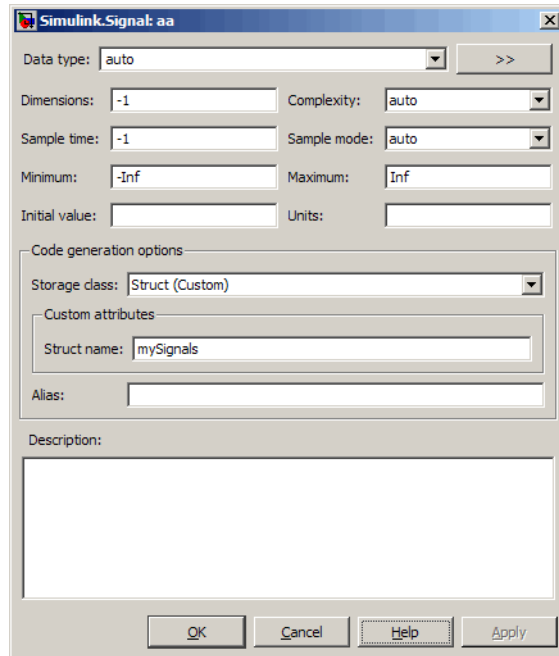
```
aa = Simulink.Signal  
bb = Simulink.Signal  
cc = Simulink.Signal
```

Alternatively, you can create the signal objects in the Simulink Model Explorer by clicking **Add Simulink Signal** or selecting **Add Simulink.Signal** from the **Add** menu.

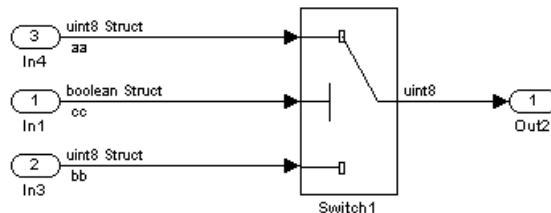
Assign Storage Class and Instance-Specific Properties

The next step is to assign the Struct custom storage class to the signal objects. The easiest way to do this is to use the object dialog box in the Model Explorer to set the RTWInfo attributes of the signal objects. The following figure illustrates how to set the **Storage class** and **Struct name** attributes for the signal object aa.

Signal objects bb and cc (not shown) are configured identically.



The association between identically named model signals and signal objects is formed automatically. The symbols aa, bb, and cc resolve to the signal objects aa, bb, and cc, which have custom storage class Struct. 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.



Generate Code

The model is now configured 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* declares 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;
```

This example shows the use of the `Struct` class in its default configuration. Using the Custom Storage Class Designer, you can customize the `Struct` class or any of the other predefined CSCs and tailor code generation to your own requirements.

Creating Packages with CSC Definitions

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

- 1 Open the Simulink Data Class Designer by typing the following command 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.

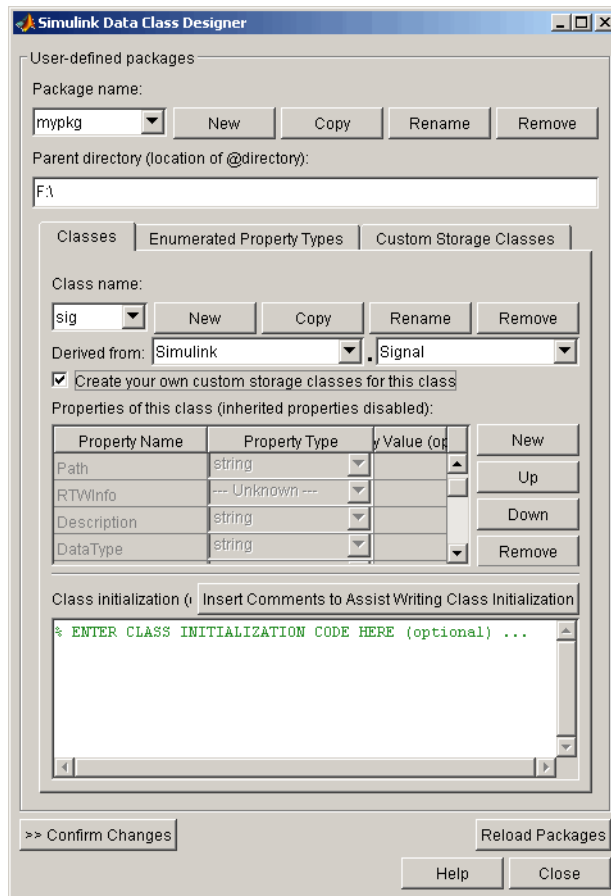
Note Do not create class package directories under *matlabroot*. Packages in these directories are treated as built-in and will not be visible in the Data Class Designer.

- 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.

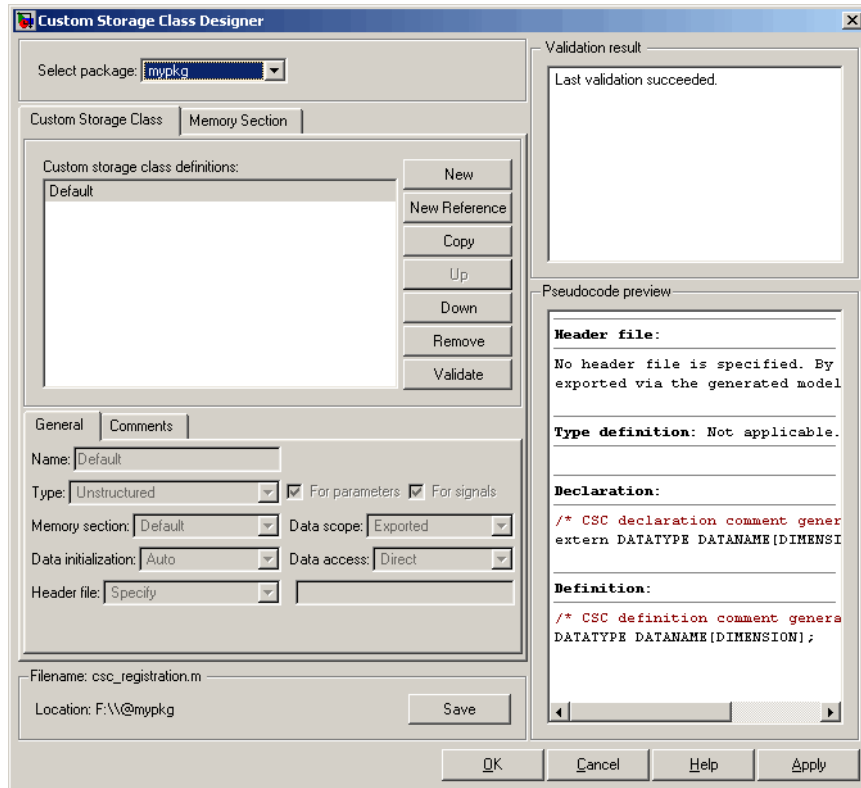


- 9 If desired, repeat steps 6–8 to add other derived classes to the package and associate CSCs with them.
- 10 Click **Confirm Changes**. In the **Confirm Changes** pane, select the package you created. Add the parent directory to the MATLAB path if necessary. Then, click **Write Selected**.

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 6-20. 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 6-20
“Editing Custom Storage Class Properties” on page 6-25
“Using Custom Storage Class References” on page 6-32
“Editing Memory Section Definitions” on page 6-35
“Using Memory Section References” on page 6-38

Using the Custom Storage Class Designer

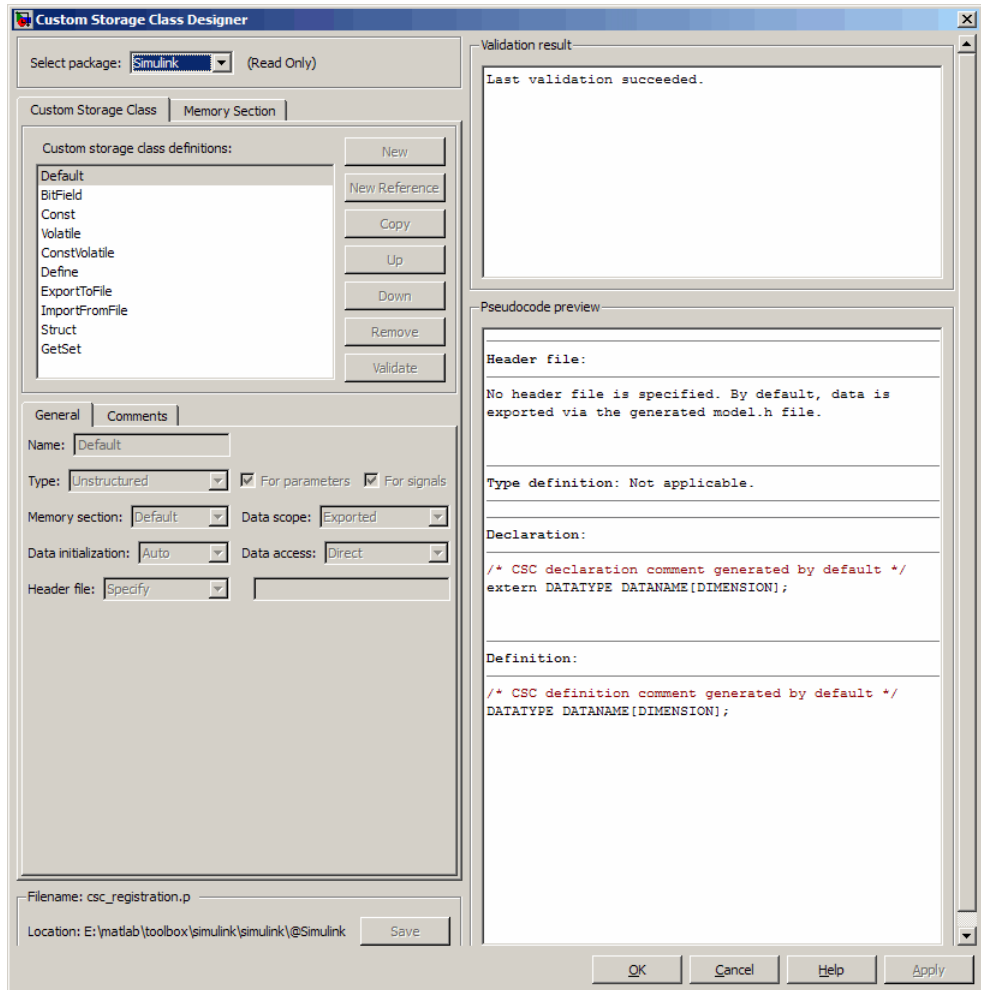
The Custom Storage Class Designer (`cscdesigner`) is a tool for creating and managing custom storage classes and memory sections. You can use the Custom Storage Class Designer to:

- 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:

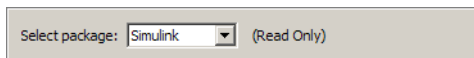


The Custom Storage Class Designer window is divided into several panels:

- **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 6-22 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 6-23.
 - 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 6-25 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 “Editing Memory Section Definitions” on page 6-35 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 6-24 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 6-37 for details.
- **Validation result:** Displays any errors encountered when the currently selected CSC definition is validated. See “Validating CSC Definitions” on page 6-32 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 and displays the associated package names in the **Select package** panel:

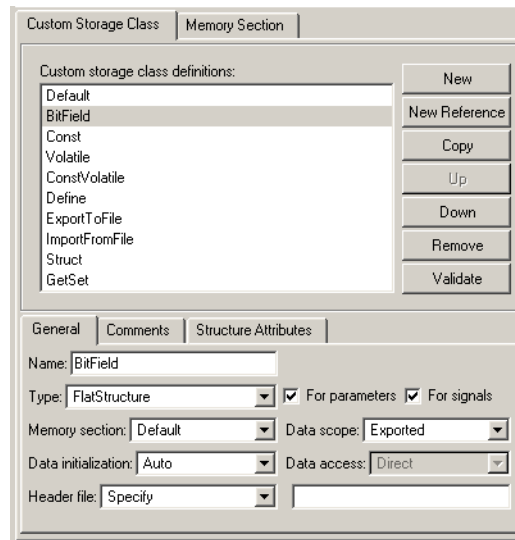


The **Select package** panel contains a menu of names of all data class packages that have a CSC registration file on the MATLAB search path. At least one such package, the Simulink package, is always present.

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.

Selecting and Editing CSCs, Memory Sections, and References

The **Custom Storage Class / Memory Section** panel lets you select, view, and — if the CSC is writeable — edit CSC or memory section definitions and references. In the picture below, the **Custom Storage Class** tab is selected.



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 6-25 for information about the properties of the predefined CSCs.
- “Editing Memory Section Definitions” on page 6-35 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 6-32 and “Using Memory Section References” on page 6-38.
- **Copy:** Creates a copy of the selected definition / reference. Copies are given a default name using the convention:

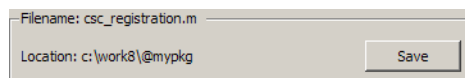
`definition_name_n`

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.

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. These properties are shown in the next figure.

The screenshot shows a dialog box with three tabs: 'General', 'Comments', and 'Structure Attributes'. The 'General' tab is active. It contains the following fields and options:

- Name:** BitField
- Type:** FlatStructure (dropdown menu)
- For parameters:** (checkbox)
- For signals:** (checkbox)
- Memory section:** Default (dropdown menu)
- Data scope:** Exported (dropdown menu)
- Data initialization:** Auto (dropdown menu)
- Data access:** Direct (dropdown menu)
- Header file:** Specify (dropdown menu)

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 6-30.
 - **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 6-42 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 “Editing Memory Section Definitions” on page 6-35.

- **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** 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 its **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 and definitions 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.

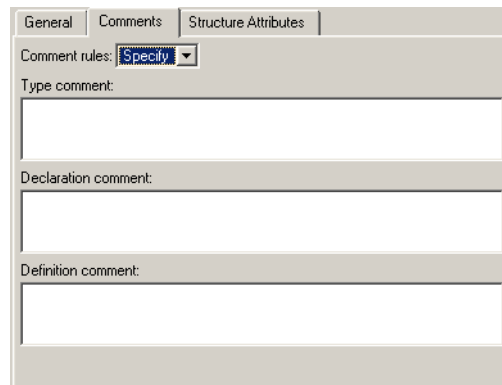
If the **Data scope** is **Exported**, specifying a header file is optional. If you specify a header file name, the custom storage class generates a header file

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 ANSI⁶ C standard (`/*...*/`). Use `\n` to specify a new line.

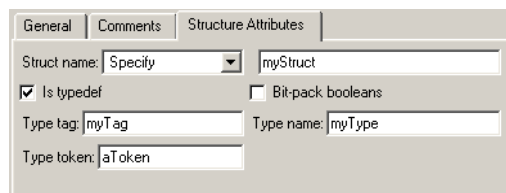
Properties in the **Comments** panel are as follows:

- **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.

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

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 6-30) 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 6-30.

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;
```

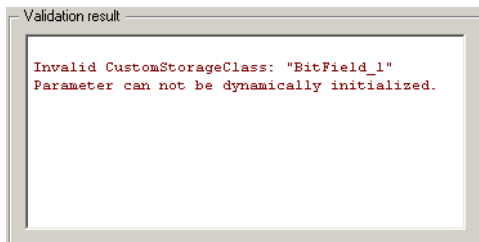
Definition:

```
/* 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:



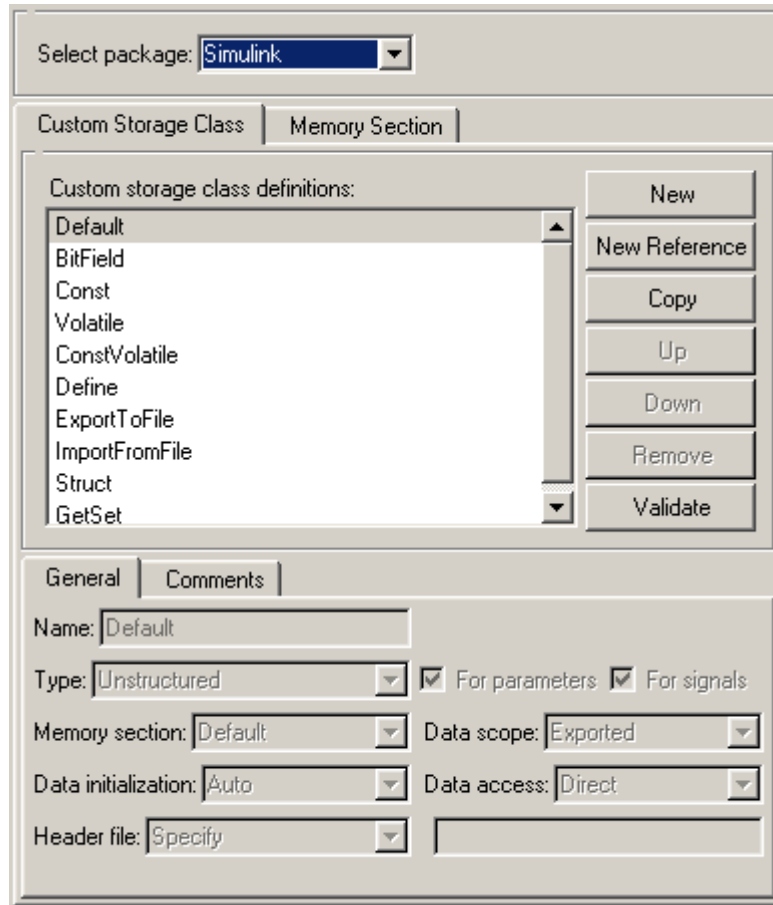
Validation is also performed whenever CSC definitions are saved. In this case, all CSC definitions are selected. (See “Saving Your Definitions” on page 6-24.)

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.

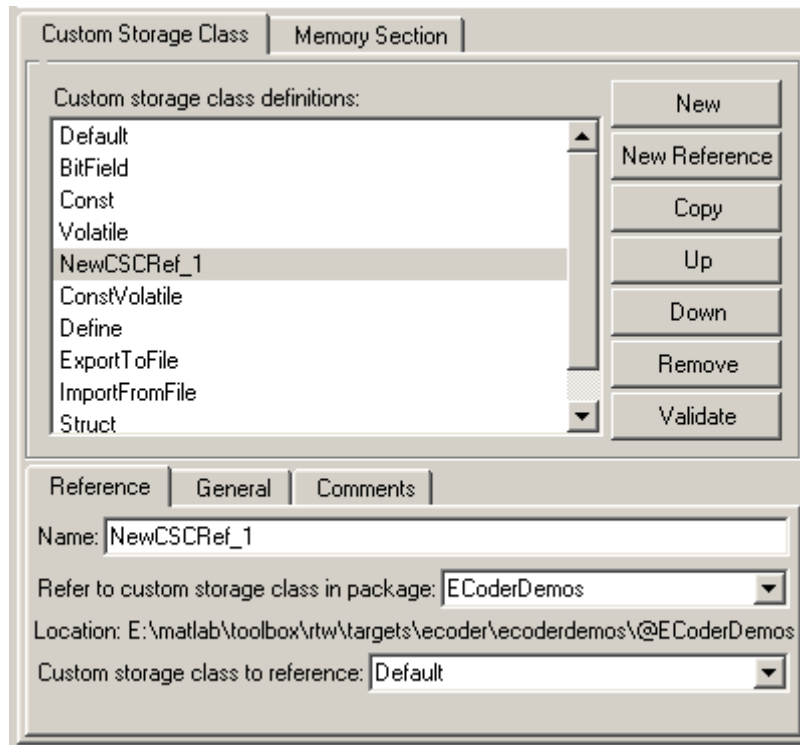
To configure a package to use a custom storage class that is defined in another package:

- 1 Type `cscdesigner` to launch the Custom Storage Class Designer. The relevant part of the designer window 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.
- 4** In the **Custom storage class 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. 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 6-24 for information about saving changes permanently.

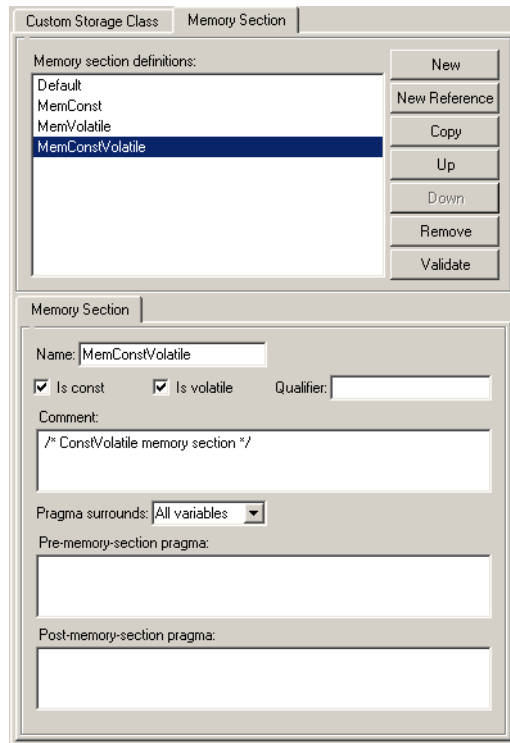
You can use Custom Storage Class Designer capabilities to copy, reorder, validate, and otherwise manage classes that have been added to a class by reference. However, you cannot change the underlying definitions. You can change a custom storage class only in the package where it was originally defined.

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 6-24 for information about saving changes permanently.

Editing Memory Section Definitions

The **Memory Section** panel lets you view, edit, and verify memory section definitions. Memory section definitions add comments, qualifiers, and `#pragma` directives to generated symbol declarations. The next figure shows the **Memory Section** panel with the `MemConstVolatile` memory section selected:



The **Memory section definitions** list lets you select a memory section definition to view or edit. The predefined memory section definitions are as follows:

- **Default:** A placeholder definition (read-only).
- **MemConst:** Generates a `const` declaration.
- **MemVolatile:** Generates a `volatile` declaration.
- **MemConstVolatile:** Generates a `const volatile` declaration.

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. The relevant part of the designer window looks like this:

Select package: **Simulink**

Custom Storage Class | **Memory Section**

Memory section definitions:

Default	New
MemConst	New Reference
MemVolatile	Copy
MemConstVolatile	Up
	Down
	Remove
	Validate

Memory Section

Name:

Is const Is volatile Qualifier:

Comment:

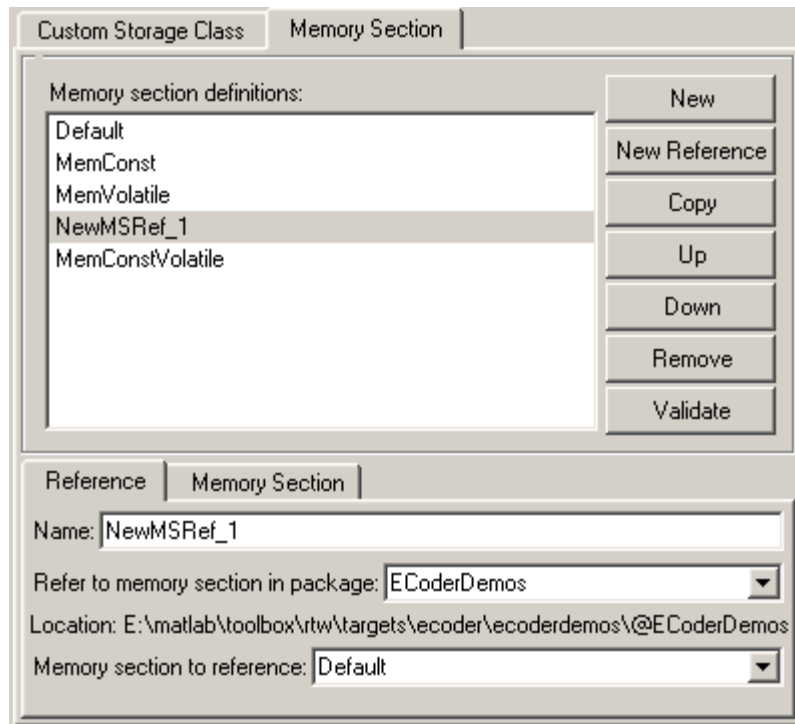
Pragma surrounds: **All variables**

Pre-memory-section pragma:

Post-memory-section pragma:

- 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. 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.

- 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 6-24 for information about saving changes permanently.

You can use Custom Storage Class Designer capabilities to copy, reorder, validate, and otherwise manage 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 6-24 for information about saving changes permanently.

Defining Advanced Custom Storage Class Types

In this section...
“Overview” on page 6-42
“Create Your Own Parameter and Signal Classes” on page 6-42
“Create a Custom Attributes Class for Your CSC (Optional)” on page 6-43
“Write TLC Code for Your CSC” on page 6-43
“Register Custom Storage Class Definitions” on page 6-44

Overview

Certain data layouts (for example, nested structures) cannot be generated using the standard `Unstructured` and `FlatStructure` custom storage class types. You can create a new custom storage class from scratch if you want to generate other types of data. Note that this requires knowledge of TLC programming and use of a special advanced mode of the Custom Storage Class Designer.

The `GetSet` CSC (see “Creating Packages with CSC Definitions” on page 6-16) is an example of an advanced CSC that is provided with the Real-Time Workshop Embedded Coder software.

The following 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 with CSC Definitions” on page 6-16.

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 “Predefined Custom Storage Classes” on page 6-6 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 `t1c` directory inside your package’s `@directory` (if it does not already exist). The naming convention to follow is

```
@PackageName/t1c
```

- 2 Copy `TEMPLATE_v1.t1c` (or another CSC template) from `matlabroot/toolbox/rtw/targets/ecoder/csc_templates` into your `t1c` 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.

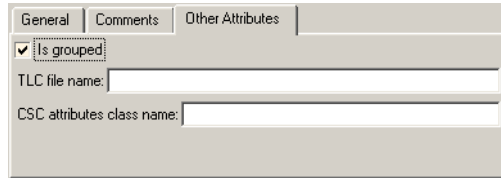
The advanced mode supports selection of an additional storage class **Type**, designated **Other**. The **Other** type is designed to support special CSC types that cannot be accommodated by the standard **Unstructured** and **FlatStructure** custom storage class types. The **Other** type cannot be assigned 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 `/tlc` 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 6-43).
- 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 6-46

“Example of Generated Code with GetSet Custom Storage Class” on page 6-48

Overview

The GetSet custom storage class is designed to generate specialized function calls to read from (get) and write to (set) the memory associated with a Data Store Memory block, when there is a need to read/write a signal many times in a single model.

GetSet CSC is capable of handling signals other than data stores. GetSet is supported for the outputs of most built-in blocks provided by The MathWorks, and the set of supported blocks is ever-growing and backward compatible. However, it is not supported for user-written S-functions. A workaround is to drop a Signal Conversion block at the output of an S-Function (or unsupported built-in block) and assign the GetSet storage class to the output of the Signal Conversion block.

The instance-specific properties of the GetSet storage class are summarized in GetSet Storage Class Properties on page 6-46.

GetSet Storage Class Properties

Property	Description
GetFunction	String that specifies the name of a function call to read data.

GetSet Storage Class Properties (Continued)

Property	Description
SetFunction	String that specifies the name of a function call to write data.
HeaderFile (optional)	<p data-bbox="624 447 1332 510">String that specifies the name of a header (.h) file to add as an <code>#include</code> in the generated code.</p> <hr data-bbox="624 569 1332 571"/> <p data-bbox="624 578 1332 803">Note 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.</p> <hr data-bbox="624 812 1332 814"/>

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

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.

The following restrictions apply to the `GetSet` custom storage class:

- The `GetSet` custom storage class supports only signals of non-complex data types.
- The `GetSet` custom storage class is designed for use with the state of the Data Store Memory block

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

```
cscdesigner -advanced
```

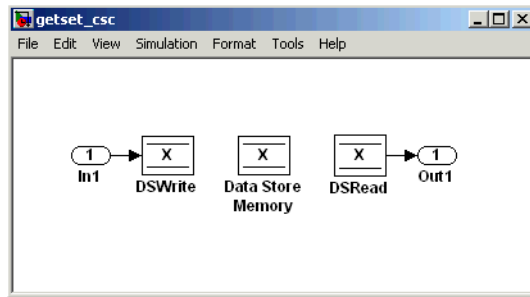
For more details about the definition of the `GetSet` storage class, look at its associated TLC code in the file

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

Example of Generated Code with `GetSet` Custom Storage Class

The model below contains a Data Store Memory that resolves to Simulink signal object `X`. `X` is configured to use the `GetSet` custom storage class as follows:

```
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 following code is generated for this 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 The Data Store Memory block creates a local variable to ensure that its value does not change in the middle of a simulation step. This also avoids multiple calls to the data's GetFunction.

Setting Code Generation Options for Custom Storage Classes

The following code generation options affect the operation of CSCs:

- During code generation, custom storage classes assigned to parameters are ignored unless the **Inline parameters** option in the **Optimization** pane of the Configuration Parameters dialog box is selected. When configuring your model and its parameters, the recommended practice is to select the **Inline parameters** option first, then assign storage classes to the desired variables or objects.

In this respect, code generation with custom storage classes behaves identically to code generation with built-in storage classes.

- Before generating code, make sure that the **Ignore custom storage classes** option in the **Custom storage classes** subpane of the **Real-Time Workshop** pane of the Configuration Parameters dialog box is deselected. When this option is selected, data objects with custom storage classes are treated as if their storage class attribute is set to Auto.

Custom Storage Class Limitations

- The Fcn block does not support parameters with custom storage class in code generation.
- For CSCs in models that use model referencing:

Note The term *grouped CSC* refers to a CSC that results in multiple data objects (in the base workspace) being referenced with a single variable in the generated code. For example, several signal objects might be grouped together in a structure by using the `Struct` or `Bitfield` custom storage classes. Data grouped in this way are referred to as *grouped data*.

- If data is assigned a grouped CSC, the CSC's **Data scope** property must be `Imported` and the data declaration must be provided in a user-supplied header file.
- If data is assigned an ungrouped CSC (for example, `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.

Older Custom Storage Classes (Prior to Release 14)

In this section...
“Introduction” on page 6-52
“Simulink.CustomParameter Class” on page 6-52
“Simulink.CustomSignal Class” on page 6-54
“Instance-Specific Attributes for Older Storage Classes” on page 6-57
“Assigning a Custom Storage Class to Data” on page 6-59
“Code Generation with Older Custom Storage Classes” on page 6-59
“Compatibility Issues for Older Custom Storage Classes” on page 6-60

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 6-60).

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 6-57.
- `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 6-57.

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 6-55, 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 const type qualifier in declaration	Y	N	any	Y	Y
ConstVolatile	Use const volatile type qualifier in declaration	Y	N	any	Y	Y
Volatile	Use volatile 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 6-55, 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 6-56, 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

BitField, Define, and Struct Storage Classes (Prior to Release 14) (Continued)

Class Name	Purpose	Parameters	Signals	Data types	Complex	Wide
Define	Represent parameters with a #define macro	Y	N	any	N	N
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 6-58 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 <i>model_export.h</i> by default.	Y
ImportFromFile	FileName	String. Defines the name of the generated header file which to be used in <code>#include</code> directive.	N
ImportFromFile	IncludeDelimiter	Enumerated. Defines delimiter used for filename in the <code>#include</code> directive. Delimiter is either double quotes (for example, <code>#include "vars.h"</code>) or angle brackets (for example, <code>#include <vars.h></code>). The default is quotes.	Y
BitField	BitFieldName	String. Defines name of bit field in which data is embedded; if unassigned, the name defaults to <code>rt_BitField</code> .	Y
Struct	StructName	String. Defines name of the struct in which data is embedded; if unassigned, the name defaults to <code>rt_Struct</code> .	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 “Working with Data” 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 with CSC Definitions” on page 6-16 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 6-61.

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:

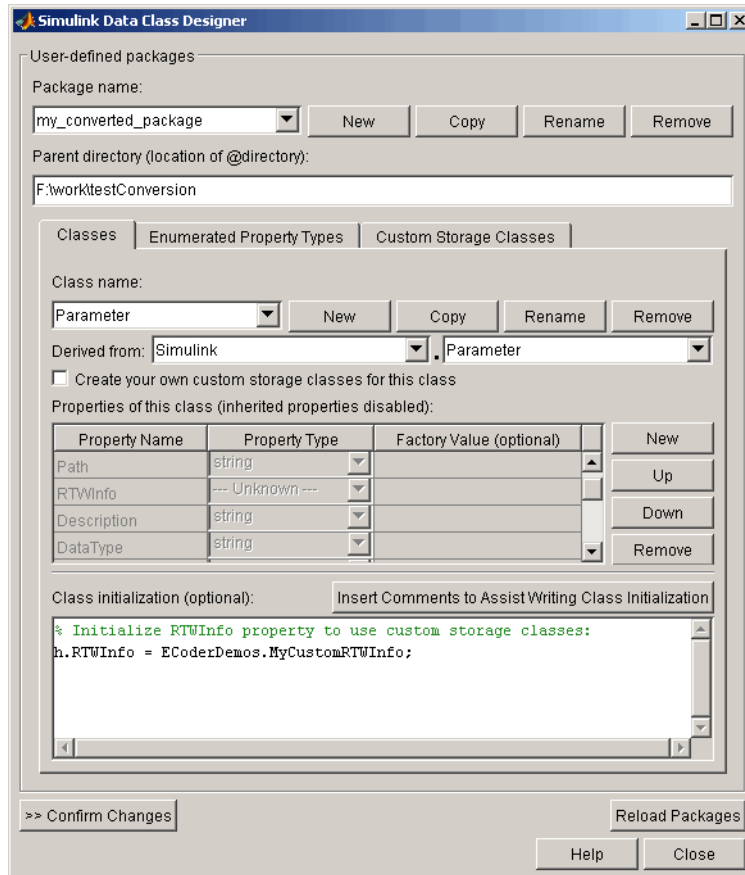
- 1 You should make a complete backup copy of the package directory before converting the package. After backing up the directory, remove the `@` prefix from the backup directory name and make sure that the backup directory is not on the MATLAB path.

- 2 Open the Simulink Data Class Designer by typing the following command at the MATLAB command prompt:

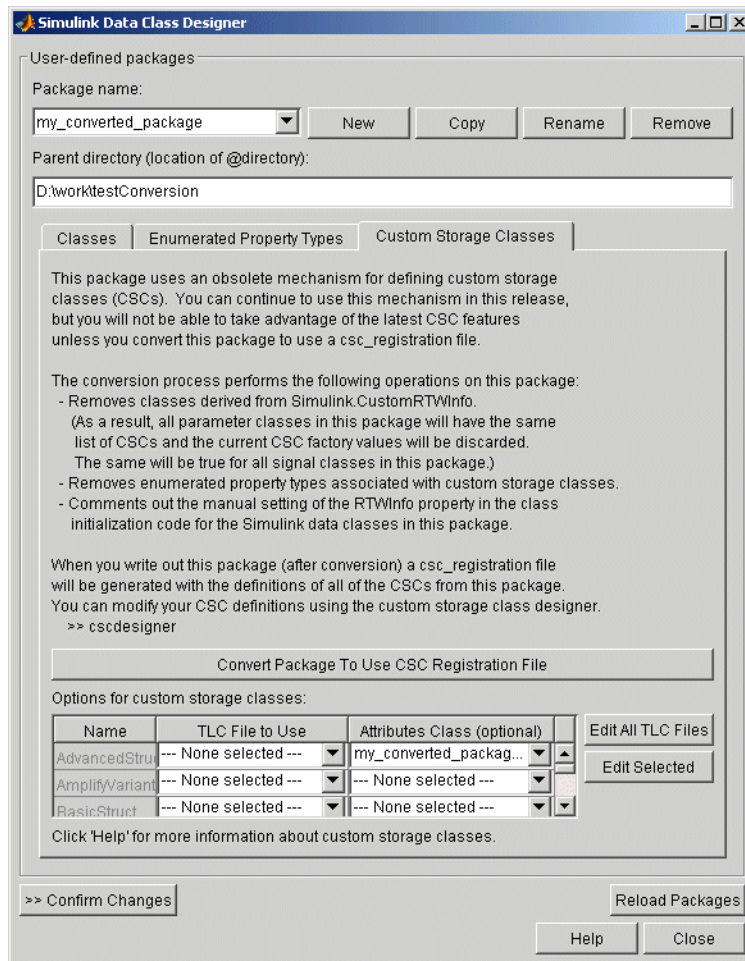
```
sldataclassdesigner
```

- 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`.



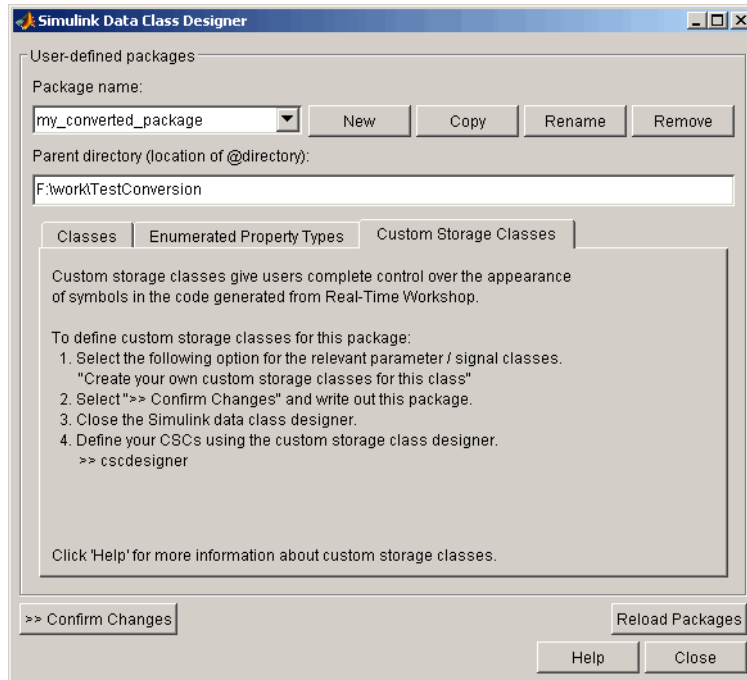
- 5 Click on the **Custom Storage Classes** pane. The pane displays a message indicating that the package contains obsolete CSC definitions, as shown in this figure.



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:

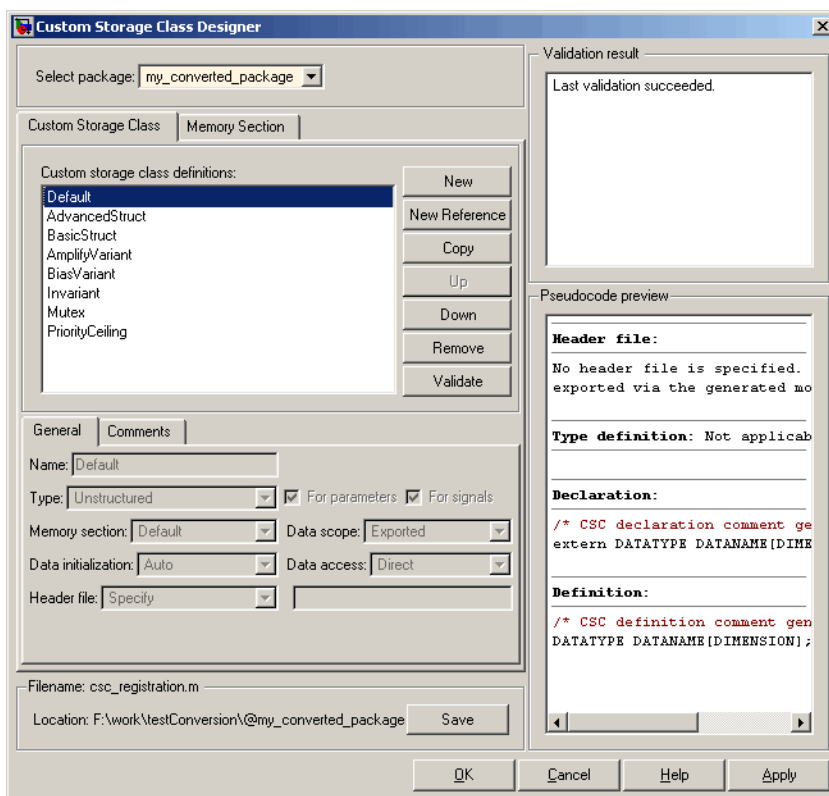
```
cscdesigner
```

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 6-20 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 6-42 for more information.

- 11** Restart your MATLAB session to ensure that your converted package is accessible.

Memory Sections

- “Introduction to Memory Sections” on page 7-2
- “Requirements for Defining Memory Sections” on page 7-4
- “Defining Memory Sections” on page 7-6
- “Applying Memory Sections” on page 7-10
- “Examples of Generated Code with Memory Sections” on page 7-18
- “Memory Section Limitation” on page 7-23

Introduction to Memory Sections

In this section...
“Overview” on page 7-2
“Memory Sections Demo” on page 7-2
“Additional Information” on page 7-2

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:

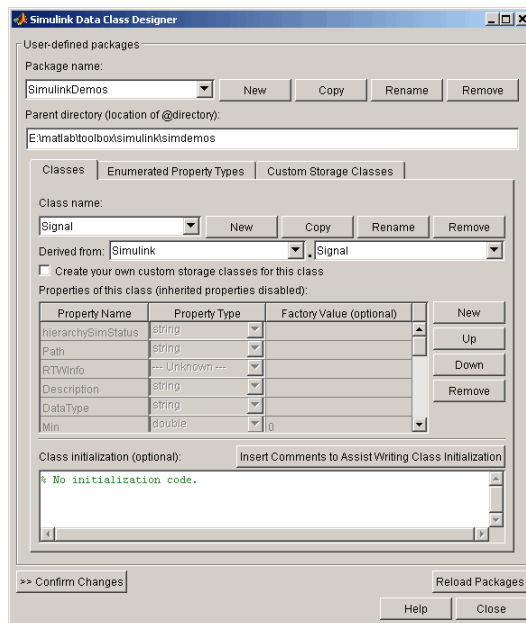
- Simulink data types, packages, data classes, and data objects:
 - “Working with Data” in the Simulink documentation
- Real-Time Workshop data structures and storage classes:
 - “Working with Data” in the Real-Time Workshop documentation
- Real-Time Workshop Embedded Coder custom storage classes:
 - Chapter 6, “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:



Instructions for using the Data Class Designer appear in “Working with Data” 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,

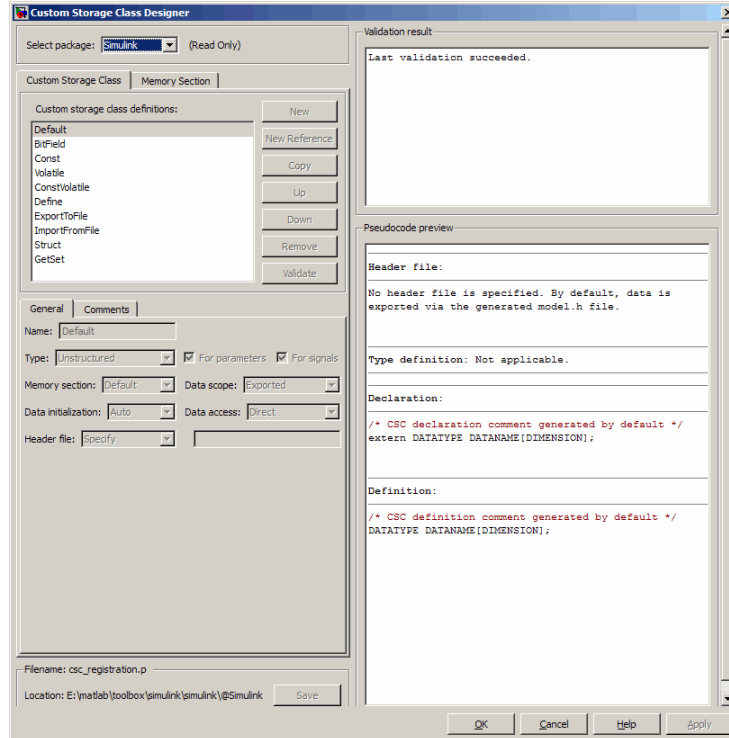
- a Choose **View > Model Explorer** in the model window.

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 looks like this:



- d Use the **Custom Storage Class** pane as needed to specify custom storage class properties. Instructions for using this pane are in “Designing Custom Storage Classes and Memory Sections” on page 6-20.

Defining Memory Sections

In this section...
“Editing Memory Section Properties” on page 7-6
“Specifying the Memory Section Name” on page 7-7
“Specifying a Qualifier for Custom Storage Class Data Definitions” on page 7-8
“Specifying Comment and Pragma Text” on page 7-8
“Surrounding Individual Definitions with Pragmas” on page 7-9
“Including Identifier Names in Pragmas” on page 7-9

Editing Memory Section Properties

After you have satisfied the requirements in “Requirements for Defining Memory Sections” on page 7-4, you can define memory sections and specify their properties. To create new memory sections or specify memory section properties,

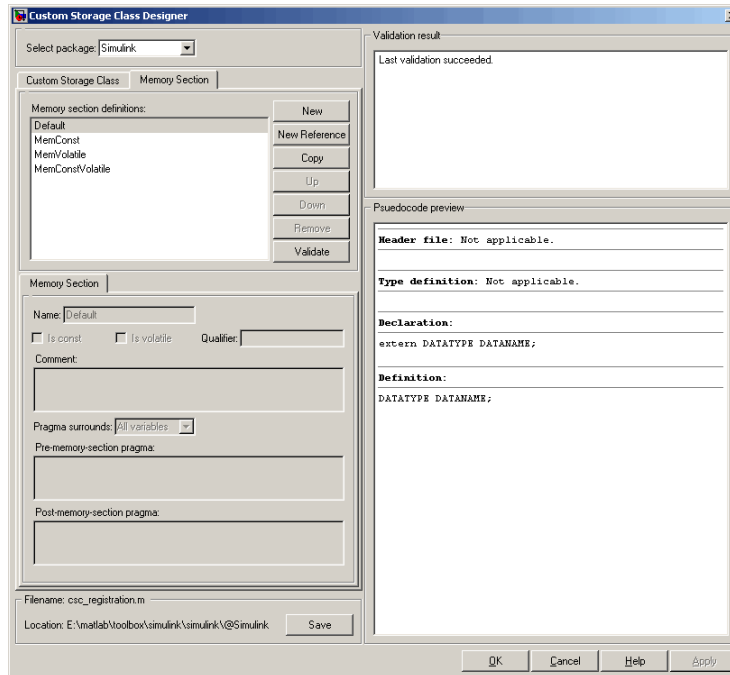
- 1 Choose **View > Model Explorer** in the model window.

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 looks like this:



The rest of this section 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 “Editing Memory Section Definitions” on page 6-35.

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, pre-pragma, or post-pragma 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 Pragas

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, pre-pragma, and post-pragma 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 pre-pragma for the section before the first definition in a contiguous block of custom storage class data definitions, and the post-pragma 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 Pragas

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 pre-pragma, the code generator will substitute the identifier from the subsequent function or data definition.
- When `%<identifier>` appears in a post-pragma, 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.

Applying Memory Sections

In this section...
“Assigning Memory Sections to Custom Storage Classes” on page 7-10
“Applying Memory Sections to Model-Level Functions and Internal Data” on page 7-12
“Applying Memory Sections to Atomic Subsystems” on page 7-15

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.

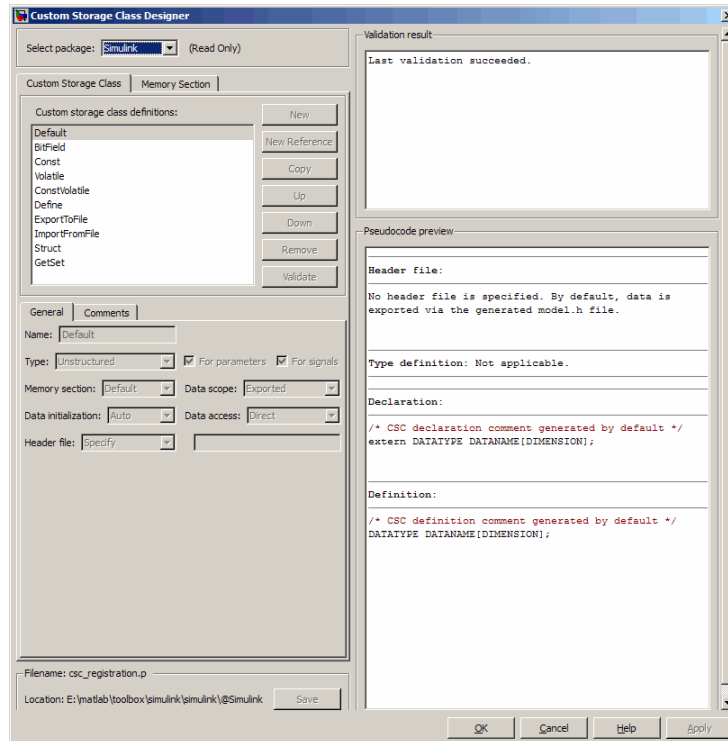
The Model Explorer appears.

- 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 looks like this:



- 4 Select the desired custom storage class in the **Custom storage class definitions** pane.
- 5 Select the desired memory section from the **Memory section** pull-down.
- 6 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 7-9. For more information, see “Creating Packages with CSC Definitions” on page 6-16.

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:

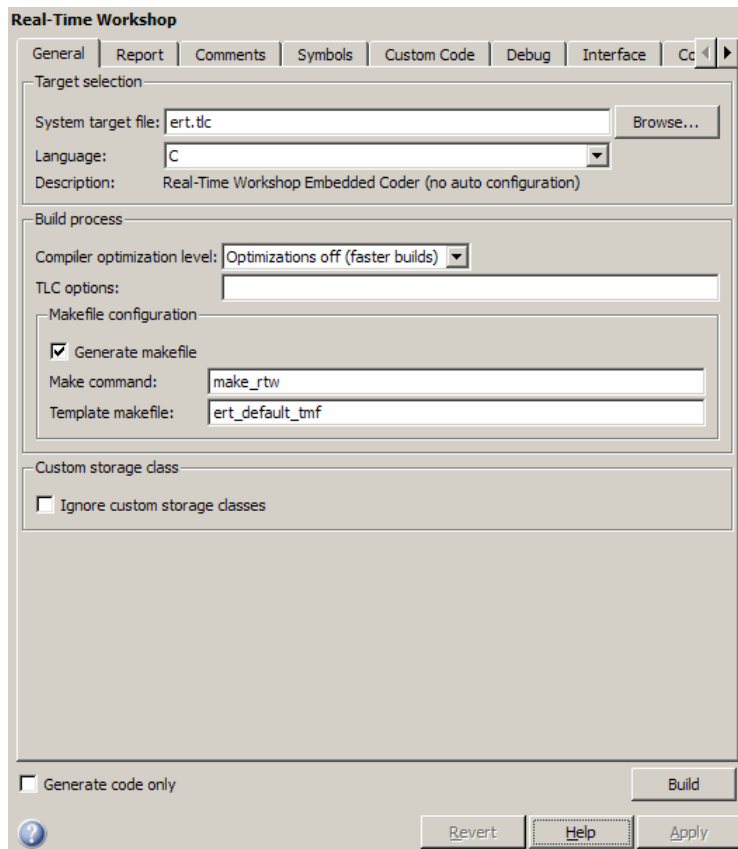
Data Category	Data Definition	Data Purpose
Constants	<i>model_cP</i>	Constant parameters
	<i>model_cB</i>	Constant block I/O
	<i>model_Z</i>	Zero representation
Input/Output	<i>model_U</i>	Root inputs
	<i>model_Y</i>	Root outputs
Internal data	<i>model_B</i>	Block I/O
	<i>model_D</i>	D-work vectors
	<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

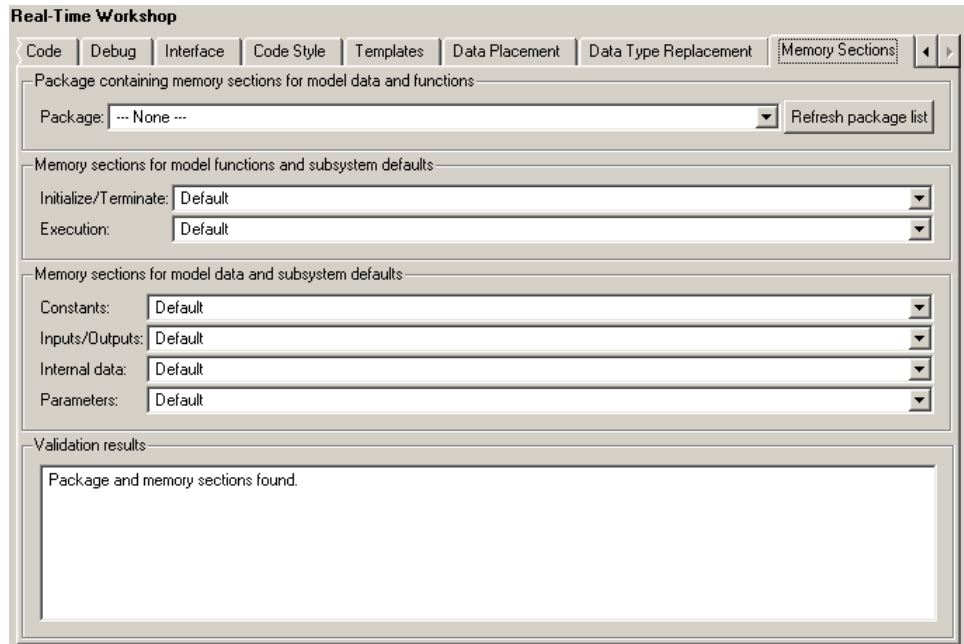
subsystems that contain overriding memory section specifications, as described in “Applying Memory Sections to Atomic Subsystems” on page 7-15.

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 top-level model, *not* from the containing atomic subsystem.

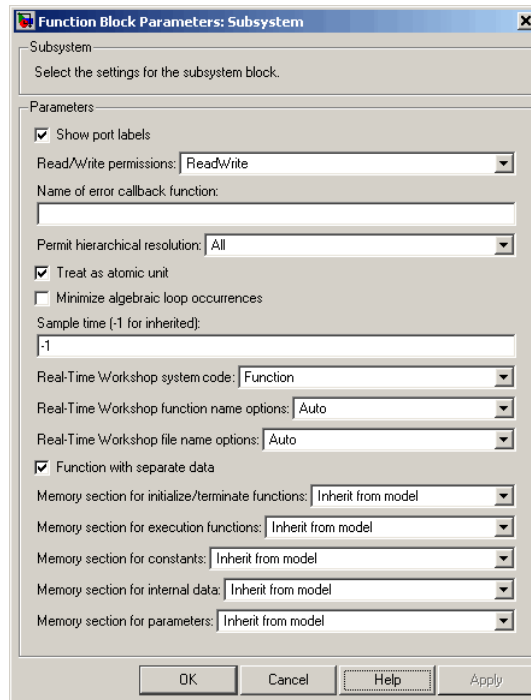
To specify memory sections for an atomic subsystem,

- 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

In this section...

“Sample ERT-Based Model with Subsystem” on page 7-18

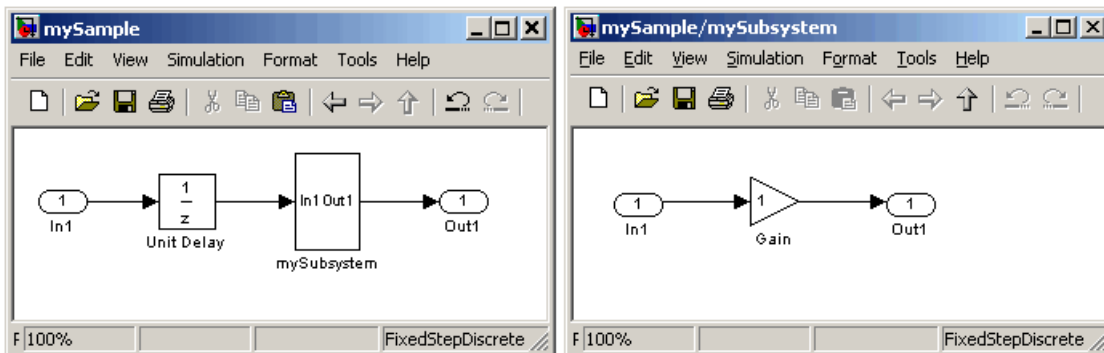
“Model-Level Data Structures” on page 7-19

“Model-Level Functions” on page 7-21

“Subsystem Function” on page 7-22

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	Pre-pragma	#pragma IO_begin
		Post-pragma	#pragma IO-end

Model-Level Memory Section Assignments and Definitions (Continued)

Section Assignment	Section Name	Field Name	Field Value
Internal data	MemSect2	Pre-pragma	#pragma InData-begin(%<identifier>)
		Post-pragma	#pragma InData-end
Parameters	MemSect3	Pre-pragma	#pragma Parameters-begin
		Post-pragma	#pragma Parameters-end
Initialize/ Terminate	MemSect4	Pre-pragma	#pragma InitTerminate-begin
		Post-pragma	#pragma InitTerminate-end
Execution functions	MemSect5	Pre-pragma	#pragma ExecFunc-begin(%<identifier>)
		Post-pragma	#pragma ExecFunc-end(%<identifier>)

Subsystem-Level Memory Section Assignments and Definitions

Section Assignment	Section Name	Field Name	Field Value
Execution functions	MemSect6	Pre-pragma	#pragma DATA_SEC(%<identifier>, "FAST_RAM")
		Post-pragma	

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 “Sharing Utility Functions”, “Supporting Shared Utility Directories in the Build Process”, and “Supporting the Shared Utilities Directory”.

Advanced Code Generation Techniques

- “Introduction” on page 8-2
- “Code Generation with User-Defined Data Types” on page 8-4
- “Customizing the Target Build Process with the STF_make_rtw Hook File” on page 8-7
- “Customizing the Target Build Process with sl_customization.m” on page 8-14
- “Auto-Configuring Models for Code Generation” on page 8-19
- “Generating Efficient Code with Optimized ERT Targets” on page 8-23
- “Custom File Processing” on page 8-31
- “Optimizing Your Model with Configuration Wizard Blocks and Scripts” on page 8-59
- “Replacement of STF_rtw_info_hook Mechanism” on page 8-72
- “Optimizing Task Scheduling for Multirate Multitasking Models on RTOS Targets” on page 8-73

Introduction

This chapter describes advanced code generation features and techniques supported by the Real-Time Workshop Embedded Coder software. These features fall into several categories:

- *User-defined data types*: How to use `Simulink.NumericType`, `Simulink.StructType` and other data type objects to map your own data type definitions to Simulink built-in data types.
- *Model configuration*: Several sections describe features that support automatic (as opposed to manual) configuration of model options for code generation. The information in each of these sections builds upon the previous section.
 - “Customizing the Target Build Process with the `STF_make_rtw` Hook File” on page 8-7 describes the general mechanism for adding target-specific customizations to the build process.
 - “Auto-Configuring Models for Code Generation” on page 8-19 shows how to use this mechanism (along with supporting utilities) to set model options affecting code generation automatically.
 - A similar mechanism is used by two special versions of the ERT target, optimized for fixed-point and floating-point code generation. These are described in “Generating Efficient Code with Optimized ERT Targets” on page 8-23.
 - “Optimizing Your Model with Configuration Wizard Blocks and Scripts” on page 8-59 describes a simpler approach to automatic model configuration. A library of Configuration Wizard blocks and scripts is provided to let you configure models quickly for common scenarios; you can also create your own scripts with minimal M-file programming.
- *Custom code generation*: These features let you directly customize generated code by creating template files that are invoked during the TLC code generation process. Basic knowledge of TLC is required to use these features.
 - “Custom File Processing” on page 8-31 describes a flexible and powerful TLC API that lets you emit custom code to any generated file (including both the standard generated model files and separate code modules).

- “Generating Custom File Banners” on page 8-53 describes a simple way to generate file banners (useful for inserting your organization’s copyrights and other common information into generated files).
- *Backward compatibility issues:* Read “Optimizing Your Model with Configuration Wizard Blocks and Scripts” on page 8-59 if you have created an `STF_rtw_info_hook` file for use with a custom target, prior to MATLAB Release 14. The `STF_rtw_info_hook` hook file mechanism for specifying target-specific characteristics for code generation has been supplanted by the simpler and more powerful **Hardware Implementation** pane of the Configuration Parameters dialog box.

Code Generation with User-Defined Data Types

In this section...

“Overview” on page 8-4

“Specifying Type Definition Location for User-Defined Data Types” on page 8-5

“Using User-Defined Data Types for Code Generation” on page 8-6

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 “Data Object 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.

Customizing the Target Build Process with the `STF_make_rtw` Hook File

In this section...

“Overview” on page 8-7

“File and Function Naming Conventions” on page 8-7

“`STF_make_rtw_hook.m` Function Prototype and Arguments” on page 8-8

“Applications for `STF_make_rtw_hook.m`” on page 8-12

“Using `STF_make_rtw_hook.m` for Your Build Procedure” on page 8-13

Overview

The build process lets you supply optional hook files that are executed at specified points in the code-generation and make process. You can use hook files to add target-specific actions to the build process.

This section describes an important M-file hook, generically referred to as `STF_make_rtw_hook.m`, where *STF* is the name of a system target file, such as `ert` or `mytarget`. This hook file implements a function, `STF_make_rtw_hook`, that dispatches to a specific action, depending on the `hookMethod` argument passed in.

The build process automatically calls `STF_make_rtw_hook`, passing in the correct `hookMethod` argument (as well as other arguments described below). You need to implement only those hook methods that your build process requires.

File and Function Naming Conventions

To ensure that `STF_make_rtw_hook` is called correctly by the build process, you must ensure that the following conditions are met:

- The `STF_make_rtw_hook.m` file is on the MATLAB path.
- The filename is the name of your system target file (STF), appended to the string `_make_rtw_hook.m`. For example, if you were generating code with a custom system target file `mytarget.tlc`, you would name your

STF_make_rtw_hook.m file to *mytarget_make_rtw_hook.m*. Likewise, the hook function implemented within the file should follow the same naming convention.

- The hook function implemented in the file follows the function prototype described in the next section.

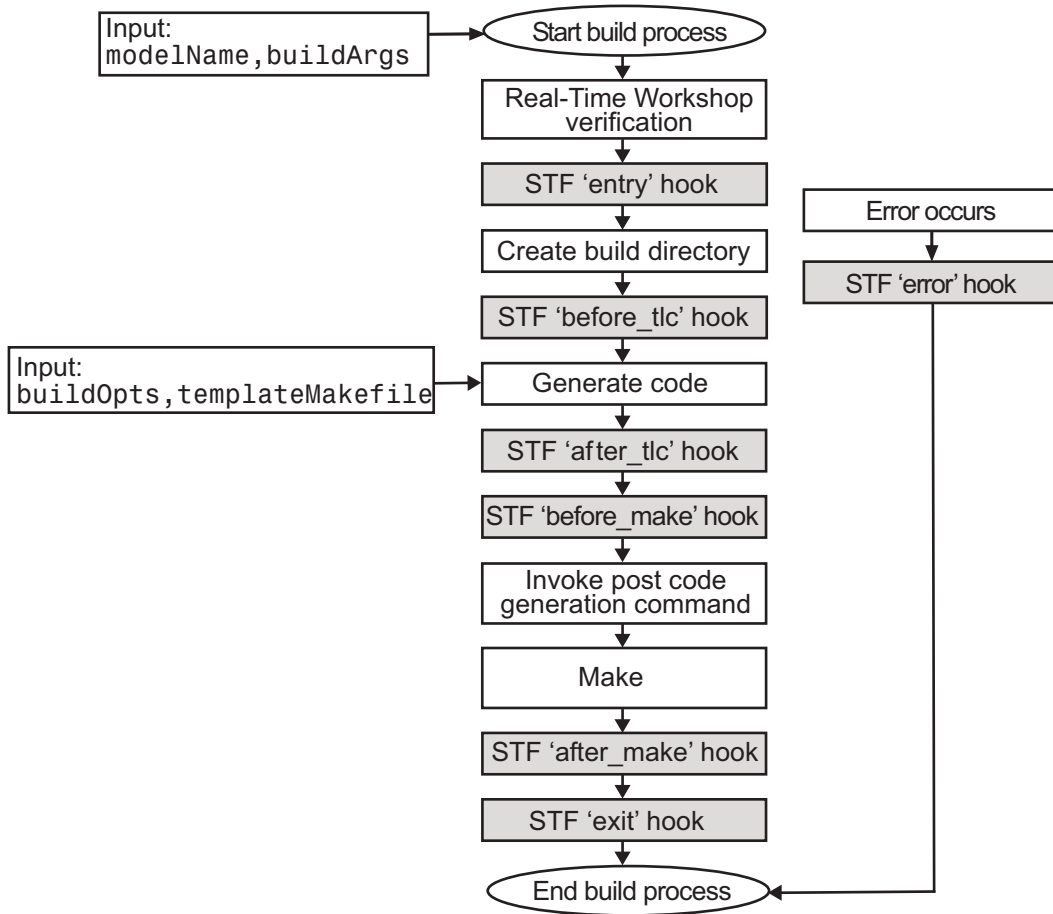
***STF_make_rtw_hook.m* Function Prototype and Arguments**

The function prototype for *STF_make_rtw_hook* is

```
function STF_make_rtw_hook(hookMethod, modelName, rtwRoot, templateMakefile,  
    buildOpts, buildArgs)
```

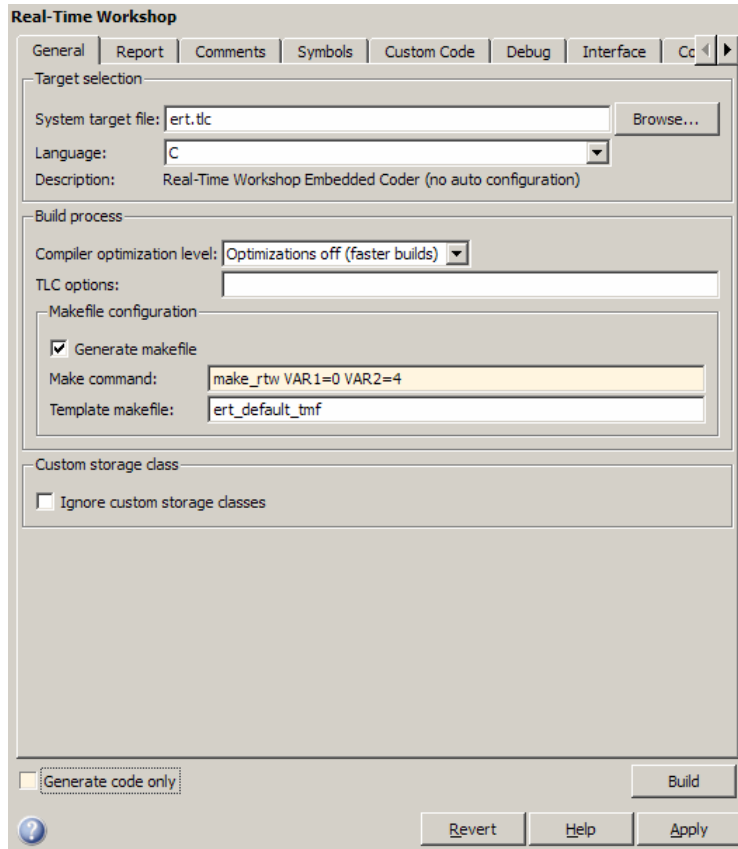
The arguments are defined as:

- *hookMethod*: String specifying the stage of build process from which the *STF_make_rtw_hook* function is called. The flowchart below summarizes the build process, highlighting the hook points. Valid values for *hookMethod* are 'entry', 'before_tlc', 'after_tlc', 'before_make', 'after_make', 'exit', and 'error'. The *STF_make_rtw_hook* function dispatches to the relevant code with a switch statement.



- modelName: String specifying the name of the model. Valid at all stages of the build process.
- rtwRoot: Reserved.
- templateMakefile: Name of template makefile.

- **buildOpts**: A MATLAB structure containing the fields described in the list below. Valid for the 'before_make', 'after_make', and 'exit' stages only. The **buildOpts** fields are
 - **modules**: Character array specifying a list of additional files that need to be compiled.
 - **codeFormat**: Character array containing code format specified for the target. (ERT-based targets must use the 'Embedded-C' code format.)
 - **noninlinedSFcns**: Cell array specifying list of noninlined S-functions in the model.
 - **compilerEnvVal**: String specifying compiler environment variable value (for example, C:\Applications\Microsoft Visual).
- **buildArgs**: Character array containing the argument to `make_rtw`. When you invoke the build process, **buildArgs** is copied from the argument string (if any) following "make_rtw" in the **Make command** field of the **Real-Time Workshop** pane of the Configuration Parameters dialog box.



The make arguments from the **Make command** field in the figure above, for example, generate the following:

```
% make -f untitled.mk VAR1=0 VAR2=4
```

Applications for `STF_make_rtw_hook.m`

An enumeration of all possible uses for `STF_make_rtw_hook.m` is beyond the scope of this document. However, this section provides some suggestions of how you might apply the available hooks.

In general, you can use the `'entry'` hook to initialize the build process before any code is generated, for example to change or validate settings. One application for the `'entry'` hook is to rerun the auto-configuration script that initially ran at target selection time to compare model parameters before and after the script executes for validation purposes.

The other hook points, `'before_tlc'`, `'after_tlc'`, `'before_make'`, `'after_make'`, `'exit'`, and `'error'` are useful for interfacing with external tool chains, source control tools, and other environment tools.

For example, you could use the `STF_make_rtw_hook.m` file at any stage after `'entry'` to obtain the path to the build directory. At the `'exit'` stage, you could then locate generated code files within the build directory and check them into your version control system. You might use `'error'` to clean up static or global data used by the hook function when an error occurs during code generation or the build process.

Note that the build process temporarily changes the MATLAB working directory to the build directory for stages `'before_make'`, `'after_make'`, `'exit'`, and `'error'`. Your `STF_make_rtw_hook.m` file should not make incorrect assumptions about the location of the build directory. You can obtain the path to the build directory anytime after the `'entry'` stage. In the following code example, the build directory pathname is returned as a string to the variable `buildDirPath`.

```
makertwObj = get_param(gcs, 'MakeRTWSsettingsObject');  
buildDirPath = getfield(makertwObj, 'BuildDirectory');
```


Using STF_make_rtw_hook.m for Your Build Procedure

To create a custom *STF_make_rtw_hook* hook file for your build procedure, copy and edit the example *ert_make_rtw_hook.m* file (located in the *matlabroot/toolbox/rtw/targets/ecoder* directory) as follows:

- 1** Copy *ert_make_rtw_hook.m* to a directory in the MATLAB path, and rename it in accordance with the naming conventions described in “File and Function Naming Conventions” on page 8-7. For example, to use it with the GRT target *grt.tlc*, rename it to *grt_make_rtw_hook.m*.
- 2** Rename the *ert_make_rtw_hook* function within the file to match the filename.
- 3** Implement the hooks that you require by adding code to the appropriate case statements within the `switch hookMethod` statement. See “Auto-Configuring Models for Code Generation” on page 8-19 for an example.

Customizing the Target Build Process with `sl_customization.m`

In this section...

“Overview” on page 8-14

“Registering Build Process Hook Functions Using `sl_customization.m`” on page 8-16

“Variables Available for `sl_customization.m` Hook Functions” on page 8-17

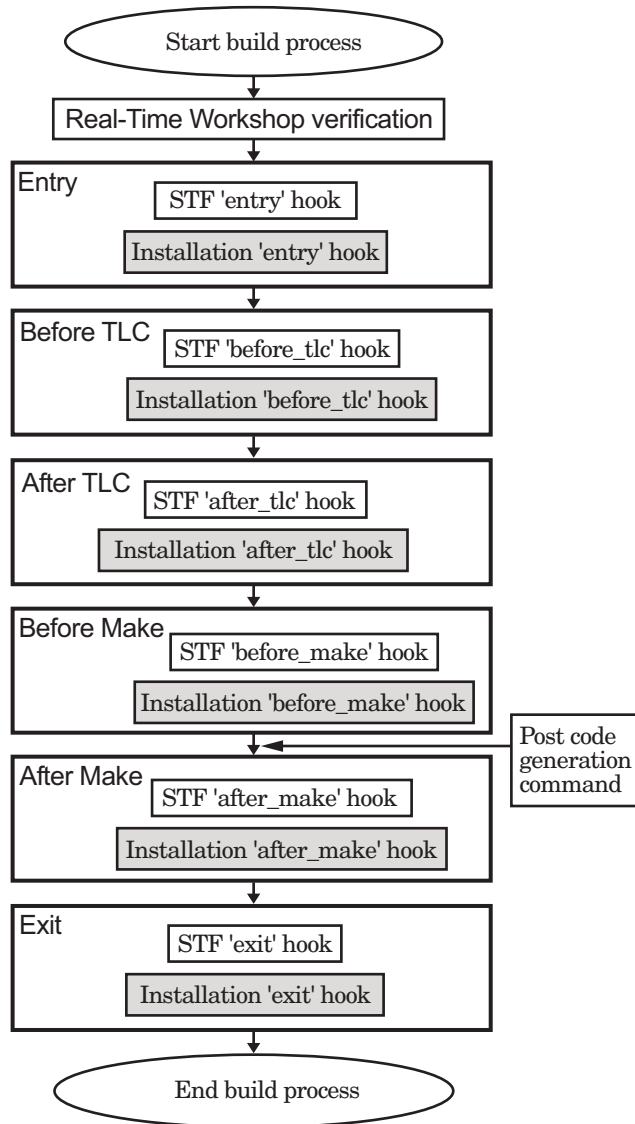
“Example Build Process Customization Using `sl_customization.m`” on page 8-17

Overview

The Simulink customization file `sl_customization.m` is a mechanism that allows you to use M-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.

The `sl_customization.m` file can be used to register installation-specific hook functions to be invoked during the Real-Time Workshop build process. The hook functions that you register through `sl_customization.m` complement System Target File (STF) hooks (described in “Customizing the Target Build Process with the `STF_make_rtw` Hook File” on page 8-7) and post-code generation commands (described in “Customizing Post Code Generation Build Processing” in the Real-Time Workshop documentation).

The following figure shows the relationship between installation-level hooks and the other available mechanisms for customizing the build process.



Registering Build Process Hook Functions Using `sl_customization.m`

To register installation-level hook functions that will be invoked during the Real-Time Workshop build process, you create an M-file function called `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function accepts one argument: a handle to an object called the `Simulink.CustomizationManager`. For example,

```
function sl_customization(cm)
```

As a starting point for your customizations, the `sl_customization` function must first get the default (factory) customizations, using the following assignment statement:

```
hObj = cm.RTWBuildCustomizer;
```

You then invoke methods to register your customizations. The customization manager object includes the following method for registering Real-Time Workshop build process hook customizations:

- `addUserHook(hObj, hookType, hook)`

Registers the hook function M-script or M-function specified by `hook` for the build process stage represented by `hookType`. The valid values for `hookType` are 'entry', 'before_tlc', 'after_tlc', 'before_make', 'after_make', and 'exit'.

Your instance of the `sl_customization` function should use this method to register installation-specific hook functions.

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, you must restart the Simulink session or enter the following command at the MATLAB command line to effect the changes:

```
sl_refresh_customizations
```

Variables Available for `sl_customization.m` Hook Functions

The following variables are available for `sl_customization.m` hook functions to use:

- `modelName` — The name of the Simulink model (valid for all stages)
- `dependencyObject` — An object containing the dependencies of the generated code (valid only for the `'after_make'` stage)

If a hook is an M-script, it can directly access the valid variables. If a hook is an M-function, it can pass the valid variables as arguments to the function. For example:

```
hObj.addUserHook('after_make', 'afterMakeFunction(modelName,dependencyObject);');
```

Example Build Process Customization Using `sl_customization.m`

The `sl_customization.m` file shown in Example 1: `sl_customization.m` for Real-Time Workshop® Build Process Customizations on page 8-17 uses the `addUserHook` method to specify installation-specific build process hooks to be invoked at the `'entry'` and `'after_tlc'` stages of the Real-Time Workshop build. For the hook function source code, see Example 2: `CustomRTWEntryHook.m` on page 8-18 and Example 3: `CustomRTWPostProcessHook.m` on page 8-18.

Example 1: `sl_customization.m` for Real-Time Workshop Build Process Customizations

```
function sl_customization(cm)
% Register user customizations

% Get default (factory) customizations
hObj = cm.RTWBuildCustomizer;

% Register Real-Time Workshop build process hooks
hObj.addUserHook('entry', 'CustomRTWEntryHook(modelName);');
hObj.addUserHook('after_tlc', 'CustomRTWPostProcessHook(modelName);');

end
```

Example 2: CustomRTWEntryHook.m

```
function [str, status] = CustomRTWEntryHook(modelName)
str =sprintf('Custom entry hook for model '%s.',modelName);
disp(str)
status =1;
```

Example 3: CustomRTWPostProcessHook.m

```
function [str, status] = CustomRTWPostProcessHook(modelName)
str =sprintf('Custom post process hook for model '%s.',modelName);
disp(str)
status =1;
```

If you include the above three files on the MATLAB path of the Simulink installation that you want to customize, the coded hook function messages will appear in the displayed output for Real-Time Workshop builds. For example, if you open the ERT-based model `rtwdemo_udt`, open the **Real-Time Workshop** pane of the Configuration Parameters dialog box, and click the **Build** button to initiate a Real-Time Workshop build, the following messages are displayed:

```
>> rtwdemo_udt

### Starting Real-Time Workshop build procedure for model: rtwdemo_udt
Custom entry hook for model 'rtwdemo_udt.'
Custom post process hook for model 'rtwdemo_udt.'
### Successful completion of Real-Time Workshop build procedure for model: rtwdemo_udt
>>
```

Auto-Configuring Models for Code Generation

In this section...

“Overview” on page 8-19

“Utilities for Accessing Model Configuration Properties” on page 8-19

“Automatic Model Configuration Using `ert_make_rtw_hook`” on page 8-20

“Using the Auto-Configuration Utilities” on page 8-22

Overview

Traditionally, model parameters are configured manually prior to code generation. It is now possible to automate the configuration of all (or selected) model parameters *at target selection time* and *at the beginning of the code generation process*. Auto-configuration is performed initially when you use the **Real-Time Workshop** pane of the Configuration Parameters dialog box to select an auto-configuration target. Auto-configuration additionally is run at the 'entry' hook point of the `STF_make_rtw_hook.m` hook file. By automatically configuring a model in this way, you can avoid manually configuring models. This saves time and eliminates potential errors. Note that you can direct the automatic configuration process to save existing model settings before code generation and restore them afterwards, so that a user's manually chosen options are not disturbed.

Utilities for Accessing Model Configuration Properties

The Simulink software provides two M-file utilities, `set_param` and `get_param` that you can use with the `STF_make_rtw_hook.m` hook file to automate the configuration of a model during the code generation process. These utilities let you configure all code-generation options relevant to the Simulink, Stateflow, Real-Time Workshop, and Real-Time Workshop Embedded Coder products. You can assign values to model parameters, backup and restore model settings, and display information about model options.

Using `set_param` to Set Model Parameters

To assign an individual model parameter value with `set_param`, pass in the model name and a parameter name/parameter value pair, as in the following examples:

```
set_param('model_name', 'SolverMode', 'Auto')
set_param('model_name', 'GenerateSampleERTMain', 'on')
```

You can also assign multiple parameter name/parameter value pairs, as in the following example:

```
set_param('model_name', 'SolverMode', 'Auto', 'RTWInlineParameters', 'off')
```

Note that the parameter names used by the `set_param` function are not always the same as the model parameter labels seen on the Configuration Parameters dialog box. For a list of parameters that you can specify and their Configuration Parameters mapping, see “Parameter Command-Line Information Summary” in the Real-Time Workshop documentation.

Automatic Model Configuration Using `ert_make_rtw_hook`

As an example of automatic model configuration, consider the example hook file, `ert_make_rtw_hook.m`. This file invokes the function `ert_auto_configuration`, which in turn calls a lower level function that sets all parameters of the model using the `set_param` utility.

While reading this section, refer to the following files, (located in `matlabroot\toolbox\rtw\targets\ecoder`):

- `ert_make_rtw_hook.m`
- `ert_auto_configuration.m`
- `ert_config_opt.m`

The `ert_config_opt` auto-configuration function is invoked first at target selection time and then again at the 'entry' stage of the build process. The following code excerpt from `ert_make_rtw_hook.m` shows how `ert_auto_configuration` is called from the 'entry' stage. At the 'exit' stage, the previous model settings are restored. Note that the `ert_auto_configuration` call is made within a `try/catch` block so that in the event of a build error, the model settings are also restored.

```
switch hookMethod
    case 'entry'
```



```

% Called at start of code generation process (before anything happens.)
% Valid arguments at this stage are hookMethod, modelName, and buildArgs.
disp(sprintf(['\n### Starting Real-Time Workshop build procedure for ', ...
            'model: %s'],modelName));

option = LocalParseArgList(buildArgs);

if ~strcmp(option,'none')
    try
        ert_unspecified_hardware(modelName);
        cs = getActiveConfigSet(modelName);
        cscopy = cs.copy;
        ert_auto_configuration(modelName,option);
        locReportDifference(cscopy, cs);
    catch
        % Error out if necessary hardware information is missing or
        % there is a problem with the configuration script.
        error(lasterr)
    end
end
...
case 'exit'
    % Called at the end of the RTW build process. All arguments are valid
    % at this stage.
    disp(['### Successful completion of Real-Time Workshop build ',...
        'procedure for model: ', modelName]);
end

```

The `ert_auto_configuration` function takes variable input arguments, the first of which is interpreted according to the type of invocation.

- The first argument is either a string specifying a model name, for 'entry' hook invocation, or a configuration set handle, for target selection invocation.
- The second argument is a string specifying a configuration mode, which is extracted from the `buildArgs` argument to `ert_make_rtw_hook.m` (see “STF_make_rtw_hook.m Function Prototype and Arguments” on page 8-8). In the example implementation, the configuration mode is either 'optimized_floating_point' or 'optimized_fixed_point'. The

following code excerpt from `ert_config_opt.m` shows a typical use of this argument to make a configuration decision:

```
if strcmp(configMode,'optimized_floating_point')
    set_param(cs,'PurelyIntegerCode','off');
elseif strcmp(configMode,'optimized_fixed_point')
    set_param(cs,'PurelyIntegerCode','on');
end
```

ert_make_rtw_hook Limitation

The code that you specify to be executed during the build process using the `ert_make_rtw_hook` mechanism cannot include a `cd` (change directory) command. For example, you cannot use `cd` in 'entry' hook code to set the build directory.

Using the Auto-Configuration Utilities

To use the auto-configuration utilities during your target selection and make processes as described above:

- 1 Set up the example `ert_make_rtw_hook.m` as your `STF_make_rtw_hook` file (see “Customizing the Target Build Process with the `STF_make_rtw_hook` File” on page 8-7).
- 2 Reconfigure the `set_param` calls within `ert_config_opt.m` to suit your application needs.

Generating Efficient Code with Optimized ERT Targets

In this section...

“Overview” on page 8-23

“Default ERT Target” on page 8-24

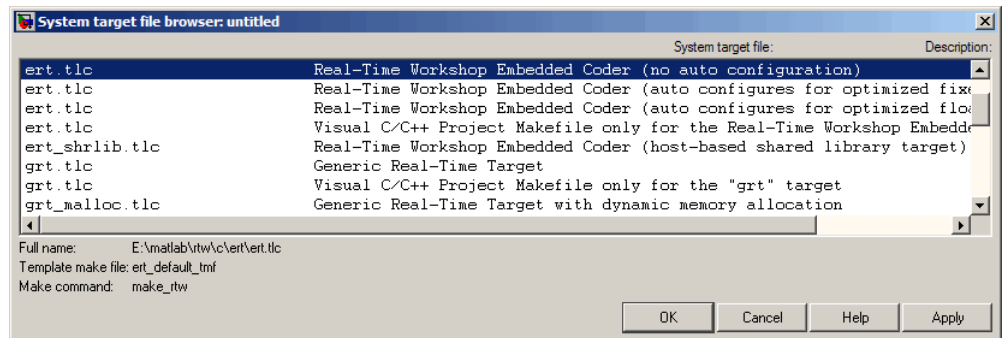
“Optimized Fixed-Point ERT Target” on page 8-24

“Optimized Floating-Point ERT Target” on page 8-26

“Using the Optimized ERT Targets” on page 8-28

Overview

To make it easier for you to generate code that is optimized for your target hardware, the Real-Time Workshop Embedded Coder software provides three variants of the 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.



The ERT target variants differ with respect to:

- Whether or not they auto-configure for optimized code generation options during the target selection and code generation processes.
- Whether or not they require specification of target hardware characteristics prior to code generation. Target hardware characteristics are configured with the options in the **Hardware Implementation** pane of the Configuration Parameters dialog box. (See “Hardware Implementation

Pane” in the Simulink documentation and “Describing Hardware Properties” in the Real-Time Workshop documentation for full details on the **Hardware Implementation** pane).

The following sections describe the ERT target variants, and how to select and use the optimized ERT targets.

Default ERT Target

The default ERT target is listed in the System Target File Browser as

```
Real-Time Workshop Embedded Coder (no auto configuration)
```

The Real-Time Workshop documentation refers to this target as the ERT target.

This target does not invoke an auto-configuration utility. Specification of target hardware characteristics is optional (although strongly recommended).

Optimized Fixed-Point ERT Target

The optimized fixed-point ERT target is listed in the System Target File Browser as

```
Real-Time Workshop Embedded Coder (auto configures for optimized fixed-point code)
```

Select this target to optimize for fixed-point code generation.

The optimized fixed-point ERT target passes in the command `optimized_fixed_point=1` to the target selection process, and also to the build process with the **Make command** field of the **Real-Time Workshop** pane of the Configuration Parameters dialog box. This in turn invokes the M-file `ert_config_opt.m`, which auto-configures the model. The auto-configuration process overrides the model settings, informing users with a message in the MATLAB command window.

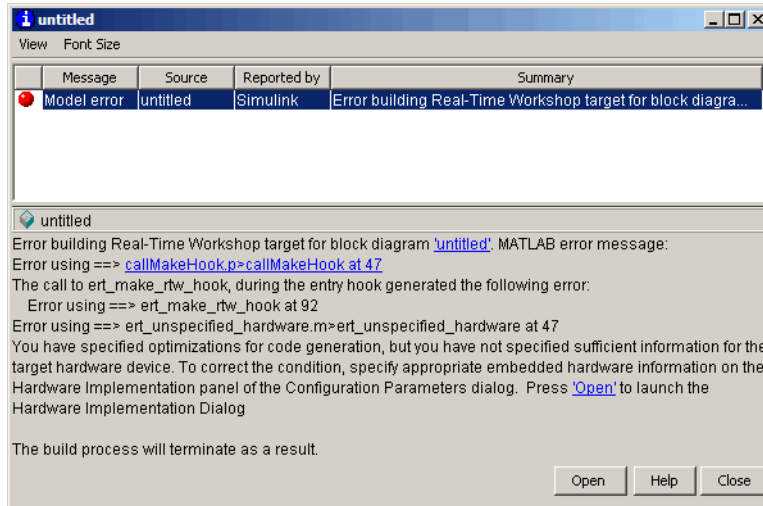
You can, if desired, customize the option settings in the auto-configuration file, `ert_config_opt.m`. See “Auto-Configuring Models for Code Generation” on page 8-19 for a complete description of the auto-configuration mechanism.

The optimized fixed-point ERT target requires specification of target hardware characteristics prior to code generation. Before generating code, you should select the desired **Device vendor** and **Device type** (or define a Custom device type with **Device vendor** set to **Generic**) in the upper panel of the **Hardware Implementation** pane of the Configuration Parameters dialog box, and set the other properties appropriately for your target. In the figure below, the **Device vendor** and **Device type** fields are configured for the Infineon C166x and XC16x microprocessor family.

The screenshot shows the 'Hardware Implementation' dialog box. It is divided into two main sections: 'Embedded hardware (simulation and code generation)' and 'Emulation hardware (code generation only)'. In the 'Embedded hardware' section, the 'Device vendor' is set to 'Infineon' and the 'Device type' is set to 'C16x, XC16x'. Below these, the 'Number of bits' section has input fields for 'char' (8), 'short' (16), 'int' (16), 'long' (32), and 'native word size' (16). The 'Byte ordering' is set to 'Little Endian' and 'Signed integer division rounds to' is set to 'Zero'. A checked checkbox indicates 'Shift right on a signed integer as arithmetic shift'. The 'Emulation hardware' section has a checked checkbox for 'None'.

Property	Value
Device vendor	Infineon
Device type	C16x, XC16x
Number of bits: char	8
Number of bits: short	16
Number of bits: int	16
Number of bits: long	32
Number of bits: native word size	16
Byte ordering	Little Endian
Signed integer division rounds to	Zero
Shift right on a signed integer as arithmetic shift	<input checked="" type="checkbox"/>
Emulation hardware (code generation only)	<input checked="" type="checkbox"/> None

If the **Device type** field is set to **Unspecified** (assume 32-bit Generic) (with **Device vendor** set to **Generic**), an error message (similar to that in the figure below) is displayed at the start of the code generation process.



Optimized Floating-Point ERT Target

The optimized floating-point ERT target is listed in the System Target File Browser as

```
Real-Time Workshop Embedded Coder (auto configures for optimized floating-point code)
```

Select this target to optimize for floating-point code generation.

The optimized floating-point ERT target passes in the command `optimized_floating_point=1` to the target selection process, and also to the build process with the **Make command** field of the **Real-Time Workshop** pane of the Configuration Parameters dialog box. This in turn invokes the M-file `ert_config_opt.m`, which auto-configures the model. The auto-configuration process overrides the model settings, informing users with a message in the MATLAB command window.

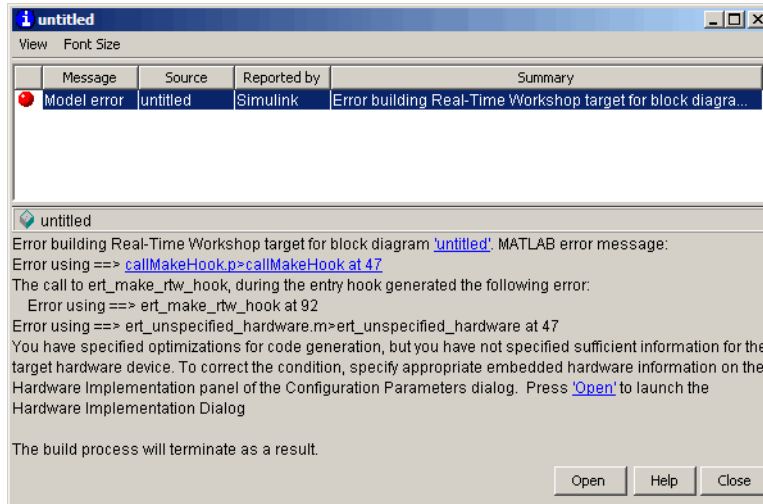
You can, if desired, customize the option settings in the auto-configuration file, file, `ert_config_opt.m`. See “Auto-Configuring Models for Code Generation” on page 8-19 for a complete description of the auto-configuration mechanism.

The optimized floating-point ERT target requires specification of target hardware characteristics prior to code generation. Before generating code, you should select the desired **Device vendor** and **Device type** (or define a Custom device type with **Device vendor** set to **Generic**) in the upper panel of the **Hardware Implementation** pane of the Configuration Parameters dialog box, and set the other properties appropriately for your target. In the figure below, the **Device vendor** and **Device type** fields are configured for the Freescale™ 32-bit PowerPC® family of microprocessors.

The screenshot shows the 'Hardware Implementation' dialog box. It is divided into two main sections: 'Embedded hardware (simulation and code generation)' and 'Emulation hardware (code generation only)'. In the 'Embedded hardware' section, 'Device vendor' is set to 'Freescale' and 'Device type' is set to '32-bit PowerPC'. Below these, there are input fields for 'Number of bits': 'char' (8), 'short' (16), 'int' (32), 'long' (32), and 'native word size' (32). 'Byte ordering' is set to 'Big Endian' and 'Signed integer division rounds to' is set to 'Zero'. A checked checkbox indicates 'Shift right on a signed integer as arithmetic shift'. In the 'Emulation hardware' section, the 'None' option is selected.

Property	Value
Device vendor	Freescale
Device type	32-bit PowerPC
Number of bits: char	8
Number of bits: short	16
Number of bits: int	32
Number of bits: long	32
Number of bits: native word size	32
Byte ordering	Big Endian
Signed integer division rounds to	Zero
Shift right on a signed integer as arithmetic shift	<input checked="" type="checkbox"/>
Emulation hardware (code generation only)	<input checked="" type="checkbox"/> None

If the **Device type** field is set to **Unspecified** (assume 32-bit Generic) (with **Device vendor** set to **Generic**), an error message (similar to that in the figure below) is displayed at the start of the code generation process.



Using the Optimized ERT Targets

This section describes how to use the optimized ERT targets in code generation.

Configuring Hardware Implementation Properties

Before using one of the optimized versions of the ERT targets, make sure that you have specified the **Hardware Implementation** properties for the model's active configuration set correctly. If this is not done properly, an error message displays at the start of the code generation process and the build terminates.

To avoid such problems, select the desired **Device vendor** and **Device type** (or define a Custom device type with **Device vendor** set to **Generic**) in the upper panel of the **Hardware Implementation** pane of the Configuration Parameters dialog box, and set the other properties appropriately for your target (see "Optimized Fixed-Point ERT Target" on page 8-24 and "Optimized

Floating-Point ERT Target” on page 8-26). Do not leave the **Device type** unspecified.

Note that if your model was created prior to MATLAB Release 14 and has not yet been updated, the **Device vendor** and **Device type** default to **Generic** and **Unspecified** (assume 32-bit **Generic**), and the **Emulation hardware** properties (in the lower section of the **Hardware Implementation** pane) are in an undefined state. This condition is indicated by the presence of a button labeled **Configure current execution hardware device**, as shown in this figure.

Hardware Implementation

Embedded hardware (simulation and code generation)

Device vendor: Generic

Device type: Unspecified (assume 32-bit Generic)

Number of bits: char: 8 short: 16 int: 32
long: 32 native word size: 32

Byte ordering: Unspecified

Signed integer division rounds to: Undefined

Shift right on a signed integer as arithmetic shift

Emulation hardware (code generation only)

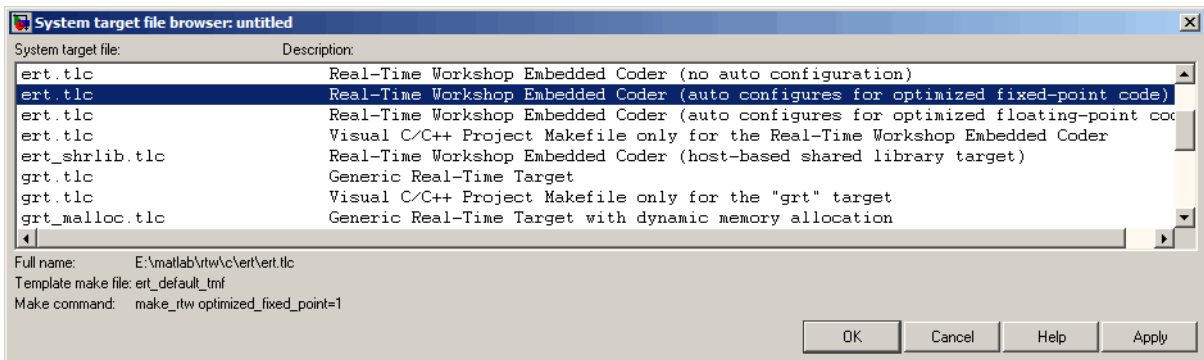
Configure current execution hardware device

In this case you should click the button to set the **Emulation hardware** properties to a valid (default) state, and save the model.

Generating Code

To generate code using one of the optimized ERT targets:

- 1 From the **Real-Time Workshop** pane of the Configuration Parameters dialog box, open the System Target File Browser and select the desired target. This figure shows the browser with the optimized fixed-point ERT target selected.



- 2 When you click **Apply** or **OK** to apply the target selection, the auto-configuration code executes. This is reported in a message similar to the following:

```
*** Auto configuring 'optimized_fixed_point' for model 'untitled' as specified by:
ert_config_opt.m
*** Overwriting model settings if they do not yield optimized code.
```

- 3 Initiate the build process.
- 4 If your model's **Hardware Implementation** parameters are not configured correctly, an error message is displayed. If the error appears, see "Configuring Hardware Implementation Properties" on page 8-28 to learn how to correct the problem, and then retry Step 3.
- 5 During code generation, the auto-configuration code executes a second time, and the auto-configuration message displayed in Step 2 appears again.
- 6 Other than the auto-configuration messages, the build process executes normally, reporting the usual progress and completion messages.

Custom File Processing

In this section...

- “Overview” on page 8-31
- “Custom File Processing Components” on page 8-32
- “Custom File Processing User Interface Options” on page 8-32
- “Code Generation Template (CGT) Files” on page 8-34
- “Using Custom File Processing (CFP) Templates” on page 8-38
- “Custom File Processing (CFP) Template Structure” on page 8-39
- “Generating Source and Header Files with a Custom File Processing (CFP) Template” on page 8-40
- “Code Template API Summary” on page 8-50
- “Generating Custom File Banners” on page 8-53

Overview

This section describes Real-Time Workshop Embedded Coder *custom file processing* (CFP) features. Custom file processing simplifies generation of custom source code by letting you

- Generate virtually any type of source (.c or .cpp) or header (.h) file. Using a *custom file processing template* (CFP template), you can control how code is emitted to the standard generated model files (for example, *model.c* or *.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 as required.
- Generate custom *file banners* (comment sections) at the start and end of generated code files.
- 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 files being generated from it.

Custom File Processing Components

The custom file processing features discussed in this section 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. CGT files are described in “Code Generation Template (CGT) Files” on page 8-34.
- *The code template API*: a high-level Target Language Compiler (TLC) API that provides functions that let you 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 useful functions. See “Code Template API Summary” on page 8-50.
- *Custom file processing (CFP) templates*: A CFP template is a TLC file that manages the process of custom code generation. The primary purpose of a CFP template is to assemble code to be generated into buffers, and to call 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 into which code is emitted. CFP templates and their applications are described in “Using Custom File Processing (CFP) Templates” on page 8-38.

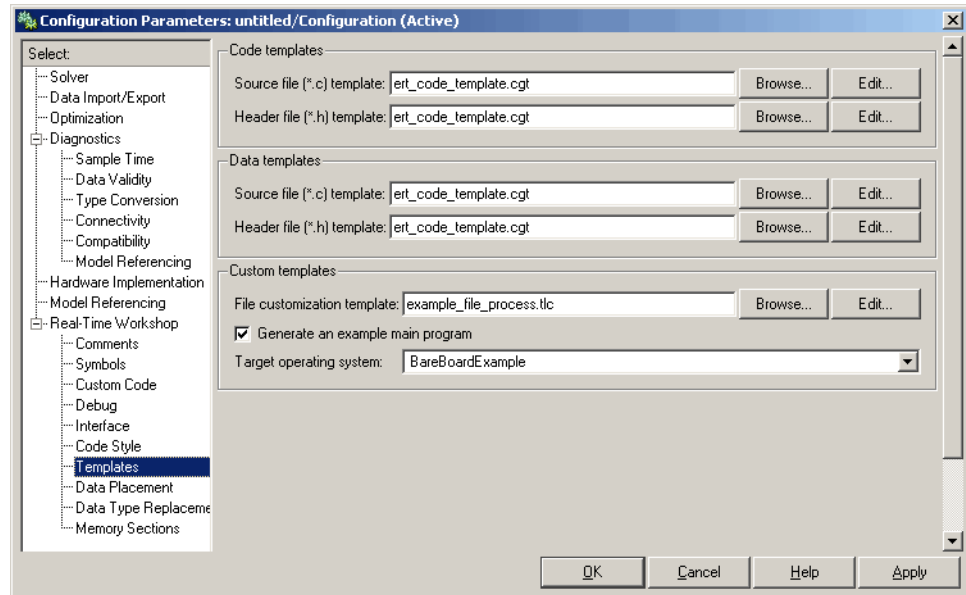
Understanding of TLC programming is required to use CFP templates. See the Target Language Compiler document to learn the basics.

Custom File Processing User Interface Options

Use of custom file processing features requires creation of CGT files and/or CFP templates. Usually, 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.

The **Templates** pane of the **Real-Time Workshop** properties of a model configuration set lets you select and edit CGT files and CFP templates, and specify their use in the code generation process. Real-Time

Workshop/Templates Pane on page 8-33 shows this pane, with 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. This file must be located 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. This file must be located 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. This file must be located on the MATLAB path. The default CFP template is `matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc`.

Each of these fields has associated **Browse** and **Edit** buttons. **Browse** lets you navigate to and select an existing CFP template or CGT file. **Edit** opens the specified CFP template into the MATLAB editor, where you can customize it.

Code Generation Template (CGT) Files

CGT files have a number of applications:

- The simplest application is generation of custom file banners (comments sections) in code files. To do this, no knowledge of the details of the CGT file structure is required; see “Generating Custom File Banners” on page 8-53.
- Some of the advanced features described in the Module Packaging Features document utilize CGT files. Refer to that document for information.
- When generating custom code using a CFP template, a CGT file is required. Correct use of CFP templates requires understanding of the CGT file structure, although in many cases it is possible to use the default CGT file without modification.

Default CGT file

The Real-Time Workshop Embedded Coder software provides a default CGT file: `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.

You should base your custom CGT files on the default file.

CGT File Structure

A CGT file consists of three sections:

Header Section. This section is optional. It contains comments and tokens for use in generating a custom header banner. “Generating Custom File Banners” on page 8-53 gives details on custom banner generation.

Code Insertion Section. This section is required. It 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

```
%<SectionName>
```

For example,

```
%<Includes>
```

The Real-Time Workshop Embedded Coder software defines a minimal set of tokens that are required for the generation of C or C++ source or header code. These are *built-in* tokens (see “Built-In Tokens and Sections” on page 8-35). You can also define *custom* tokens and add them to the code insertion section (see “Generating a Custom Section” on page 8-47).

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 on the generated code. The presence of a token in the CGT file does not guarantee that the corresponding section is generated. To generate code into a given section, you must do so explicitly by calling the code template API from a CFP template, as described in “Using Custom File Processing (CFP) Templates” on page 8-38.

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 8-37.

You can also insert C or C++ comments into the code insertion section, between tokens. Such comments are inserted directly into the generated code.

Trailer Section. This section is optional. It contains comments and tokens for use in generating a custom trailer banner. “Generating Custom File Banners” on page 8-53 gives details on custom banner generation.

Built-In Tokens and Sections

The following code extract shows the code insertion section of the default CGT file, showing the built-in tokens.

```
%% 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 carefully the following requirements before creating or customizing a CGT file:

- All the built-in tokens are required. None can be removed.
- Built-in tokens must appear in the order shown. The ordering is significant because each successive section can have dependencies on previous sections.
- Only one token can appear per line.
- Tokens must not be repeated.
- Custom tokens can be added to the code insertion section, provided that the previous requirements are not violated.
- Comments can be added to the code insertion section, provided that the previous requirements are not violated.

Built-In CGT Tokens and Corresponding Code Sections on page 8-36 summarizes the built-in tokens and corresponding section names, and describes the code sections.

Built-In CGT Tokens and Corresponding Code Sections

Token / Section Name	Description
Includes	#include directives section
Defines	#define directives section
Types	typedef section. Typedefs can depend on any previously defined type

Built-In CGT Tokens and Corresponding Code Sections (Continued)

Token / Section Name	Description
Enums	Enumerated types section
Definitions	Place data definitions here (for example, <code>double x = 3.0;</code>)
Declarations	Data declarations (for example, <code>extern double x;</code>)
Functions	C or C++ functions

Subsections

It is possible to define one or more named subsections for any section. Some of the built-in sections have predefined subsections. These are summarized in Subsections Defined for Built-In Sections on page 8-37.

It is important to note that the sections and subsections listed in Subsections Defined for Built-In Sections on page 8-37 are emitted, in the order listed, to the source or header file being generated.

The custom section feature lets you define sections in addition to those listed in Subsections Defined for Built-In Sections on page 8-37. See “Generating a Custom Section” on page 8-47 for information on how to do this.

Subsections Defined for Built-In Sections

Section	Subsections	Subsection Description
Includes	N/A	
Defines	N/A	
Types	IntrinsicTypes	Intrinsic typedef section. Intrinsic types are those that depend only on intrinsic C or C++ types.

Subsections Defined for Built-In Sections (Continued)

Section	Subsections	Subsection Description
Types	PrimitiveTypedefs	Primitive typedef section. Primitive typedefs are those that depend only on intrinsic C or C++ types and on any typedefs previously defined in the <code>IntrinsicTypes</code> section.
Types	UserTop	Any type of code can be placed in this section. You can place code that has dependencies on the previous sections here.
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	<code>#warning</code> directives
Functions	CompilerWarnings	<code>#error</code> directives
Functions	Documentation	Documentation (comment) section
Functions	UserBottom	Any code can be placed 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 8-40.

- TLC files supporting generation of single-rate and multirate main program modules (see “Customizing Main Program Module Generation” on page 8-45).

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 8-32).

Custom File Processing (CFP) Template Structure

A custom file processing (CFP) template imposes a simple structure on the code generation process. The template, in conjunction with 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 8-36 and Subsections Defined for Built-In Sections on page 8-37.

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 8-37.

Determine the `section` argument as follows:

- If Subsections Defined for Built-In Sections on page 8-37 defines no subsections for a given section, use the section name as the `section` argument.

- If Subsections Defined for Built-In Sections on page 8-37 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 8-47).
- `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 8-40, provides typical usage examples.

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, and so on 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`).

You should 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 of your choice. This directory should be located outside the MATLAB directory structure (that is, it should not be under `matlabroot`.) Note that this directory must be on the MATLAB path, or on the TLC 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
```

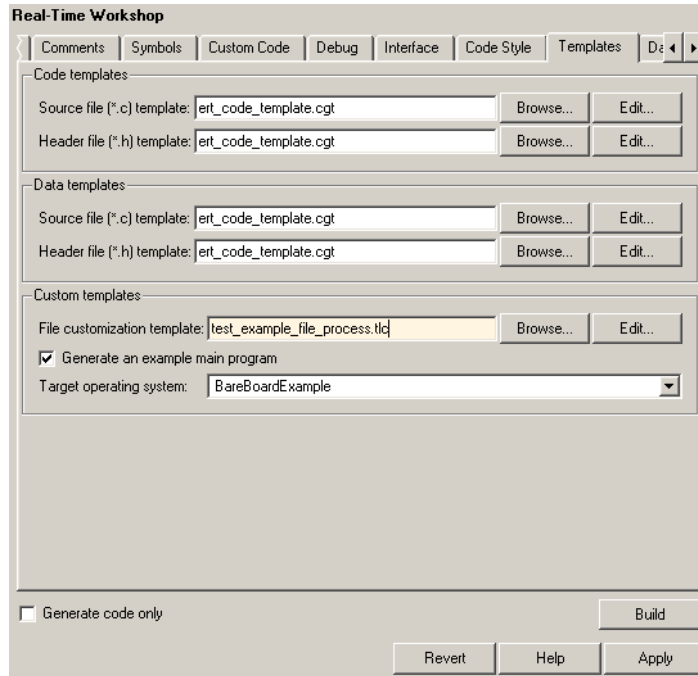
It should now read:

```
%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** properties view of the active configuration set.

- 8 Click on 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 specified as the CFP template.



- 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 Source 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. You should refer to the comments in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc` while reading the discussion below.

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.

The available file sections, and the order in which they are emitted to the generated file, are summarized in Subsections Defined for Built-In Sections on page 8-37.

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 an M 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 of the Developing Embedded Targets document 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 2-13 and “Guidelines for Modifying `rt_OneStep`” on page 2-19 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.

Code Template API Summary

Code Template API Functions on page 8-50 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 8-39).
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).

Code Template API Functions (Continued)

Function	Description
LibCallModelTerminate	Returns code for calling the model's <i>model_terminate</i> function (valid for ERT only).
LibCallSetEventForThisBaseStep	Returns code for calling the model's set events function (valid for ERT only).
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 singletasking 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).

Code Template API Functions (Continued)

Function	Description
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).
LibNumDiscreteSampleTimes	Returns the number of discrete sample times in the model.
LibSetSourceFileCodeTemplate	Set the code template to be used for generating a specified source file.
LibSetSourceFileOutputDirectory	Set the directory into which a specified source file is to be generated.
LibAddSourceFileCustomSection	Add a custom section to a source file. The custom section must be associated with one of the built-in (required) sections: Includes, Defines, Types, Enums, Definitions, Declarations, or Functions.
LibSetSourceFileCustomSection	Adds to the contents of a specified custom section within a specified file. The custom section must have been previously created with LibAddSourceFileCustomSection.
LibGetSourceFileCustomSection	Returns the contents of a specified custom section within a specified file.
LibSetCodeTemplateComplianceLevel	This function must be called from your CFP template before any other code template API functions are called. Pass in 2 as the level argument.

Generating Custom File Banners

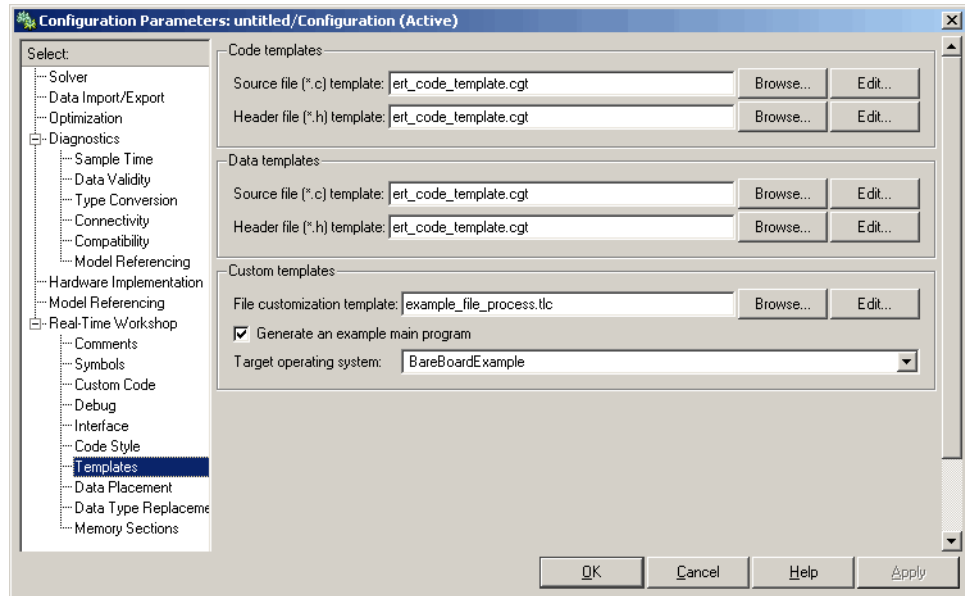
Using code generation template (CGT) files, you can specify custom *file banners* to be inserted into generated code files. File banners are comment sections in the header and trailer portions of a generated file. You can use these banners to add a company copyright statement, specify a special version symbol for your configuration management system, remove time stamps, and for many other purposes. These banners can contain non US-ASCII characters, which are propagated to the generated code.

The recommended technique for specifying file banners is to create a custom CGT file with a customized banner section. During the build process, an executable TLC file is created from the CGT file. This TLC file is then invoked during the code generation process.

You do not need to be familiar with TLC programming to generate custom banners. Generally, you simply need to modify example files supplied with the ERT target.

Note Prior releases supported direct use of customized TLC file as banner templates. These were specified with the **Source file (.c) banner template** and **Header file (.h) banner template** options of the ERT target. Direct use of a TLC file for this purpose is still supported for backward compatibility, but you should now use CGT files for this purpose instead.

File banner generation is supported by the options in the **Code templates** section of the **Templates** pane of the **Real-Time Workshop** properties of a configuration set (shown in ERT Templates Options on page 8-54).



ERT Templates Options

The options related to file banner generation are

- **Source file (.c) template:** CGT file to use when generating source (.c or .cpp) files. This file must be located on the MATLAB path.
- **Header file (.h) template:** CGT file to use when generating header (.h) files. This file must be located on the MATLAB path. This can be the same template specified in the **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*.

- Each of these fields has associated **Browse** and **Edit** buttons. **Browse** lets you navigate to and select an existing CGT file for use as a template. **Edit** opens the specified file into the MATLAB editor, where you can customize it.

Creating a Custom File Banner Template

The recommended procedure for customizing a CGT for custom file banner generation is to make a local copy of the default code template and edit it, as follows:

- 1 Activate the configuration set you want to work with.
- 2 Open the **Real-Time Workshop** properties view of the active configuration set.
- 3 Click on the **Templates** tab (see ERT Templates Options on page 8-54).
- 4 By default, the code template specified in the **Source file (.c) template** and **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, use the **Browse** button to locate and select a CGT file.
- 6 Click the **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 in conjunction with a custom target, it is good practice to locate the CGT file in a folder under your target's root directory.
- 9 It is also good practice to rename your local copy of the CGT file. When you rename the CGT file, make sure to update the associated **Source file (.c) template** or **Header file (.h) template** field to match the new filename.
- 10 Edit and customize the local copy of the CGT file for file banner generation, using the information provided in “Customizing a Code Generation Template (CGT) File for Custom Banner Generation” on page 8-56.
- 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/or header files to confirm that they contain the banners specified by the template(s).

Customizing a Code Generation Template (CGT) File for Custom Banner Generation

This section explains how to edit a CGT file for custom file banner generation. For a general description of CGT files, see “Code Generation Template (CGT) Files” on page 8-34.

To generate custom file banners, you modify a header section, a trailer section, or both, in a CGT file:

- Header section: This section contains comments and tokens for use in generating a custom header banner. The header banner precedes any C or C++ code generated by the model. If the header section is omitted, no header banner is generated. The following is the default header section provided with the default CGT file, *matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt*.

```
%% Custom file banner (optional)
%%
/*
 * 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 source code generated on    : %<SourceGeneratedOn>
 */
```

- **Trailer section:** This section contains comments and tokens for use in generating a custom trailer banner. The trailer banner follows any C or C++ code generated by the model. If the trailer section is omitted, no trailer banner is generated. The following is the default trailer section provided in the default CGT file.

```
%% Custom file trailer (optional)
%%
/* File trailer for Real-Time Workshop generated code.
 *
 * [EOF]
 */
```

The header and trailer sections typically use TLC variables (such as %<ModelVersion>) as tokens. During code generation, tokens are replaced with values in the generated code. See Summary of Tokens for File Banner Generation on page 8-58 for a list of available tokens.

The following code excerpt shows a modified banner section based on the default CGT. This template inserts a copyright notice into the banner.

```
%% Custom file banner (optional)
%%
/*
 * File: %<FileName>
 * -----
 * Copyright 2008 ABC Corporation, Inc.
 * -----
 * 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 source code generated on     : %<SourceGeneratedOn>
 *
 */
```

The following code excerpt shows a file banner generated from the `rtwdemo_u dt` model using the above template.

```

/*
 * File: rtwdemo_u dt.c
 * -----
 * Copyright 2008 ABC Corporation, Inc.
 * -----
 * Real-Time Workshop code generated for Simulink model rtwdemo_u dt.
 *
 * Model version                : 1.116
 * Real-Time Workshop file version : 7.1 (R2008a) 25-Dec-2007
 * Real-Time Workshop file generated on : Thu Jan 17 17:14:16 2008
 * TLC version                  : 7.1 (Jan 9 2008)
 * C source code generated on    : Thu Jan 17 17:14:16 2008
 *
 */

```

Summary of Tokens for File Banner Generation

FileName	Name of the generated file (for example, "rtwdemo_u dt.c").
FileType	Either "source" or "header". Designates whether generated file is a .c or .cpp file or an .h file.
FileTag	Given filenames 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.
TLCVersion	Version of Target Language Compiler.
SourceGeneratedOn	Timestamp of generated file.

Optimizing Your Model with Configuration Wizard Blocks and Scripts

In this section...

“Overview” on page 8-59

“Configuration Wizards vs. Auto-Configuring Targets” on page 8-61

“Adding a Configuration Wizard Block to Your Model” on page 8-61

“Using Configuration Wizard Blocks” on page 8-64

“Creating a Custom Configuration Wizard Block” on page 8-64

Overview

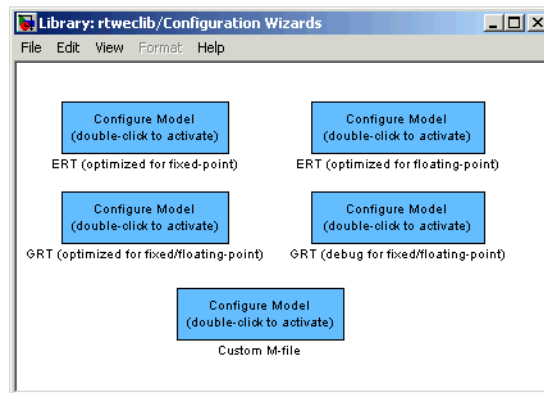
The Real-Time Workshop Embedded Coder software provides a library of *Configuration Wizard* blocks and scripts to help you configure and optimize code generation from your models quickly and easily.

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 custom M-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 M-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 M-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.

Configuration Wizards vs. Auto-Configuring Targets

Configuration Wizard scripts and auto-configuring targets offer two different approaches to automatic model configuration. You need to consider issues of complexity and the needs of your end users when choosing one or the other approach.

Auto-configuring targets (described in “Auto-Configuring Models for Code Generation” on page 8-19 and “Generating Efficient Code with Optimized ERT Targets” on page 8-23) execute a back end configuration function (hook file) during the code generation process. The auto-configuration function in effect bypasses the options set in the model’s configuration set, which are saved and restored transparently across the build process.

Configuration Wizards, on the other hand, execute a configuration script independently from the code generation process. The Configuration Wizard script actually changes the model’s active configuration set. These changes are then visible in the GUI and can be saved with the model.

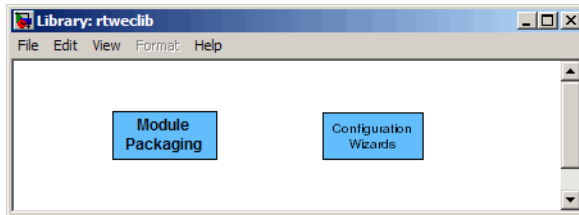
It is generally simpler to create a custom Configuration Wizard script than to create a custom auto-configuring target. Creating a Configuration Wizard script, in many cases, requires only simple modifications to an existing template. Creating a custom auto-configuring target, on the other hand, requires some knowledge of the internals of the build process.

Adding a Configuration Wizard Block to Your Model

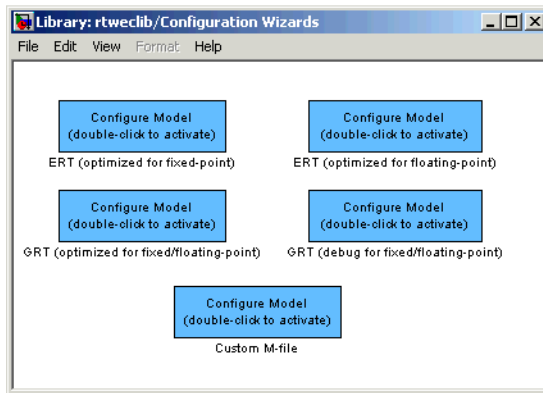
This section describes how to add one of the preset Configuration Wizard blocks to a model.

The Configuration Wizard blocks are available in the Real-Time Workshop Embedded Coder block library. To use a Configuration Wizard block:

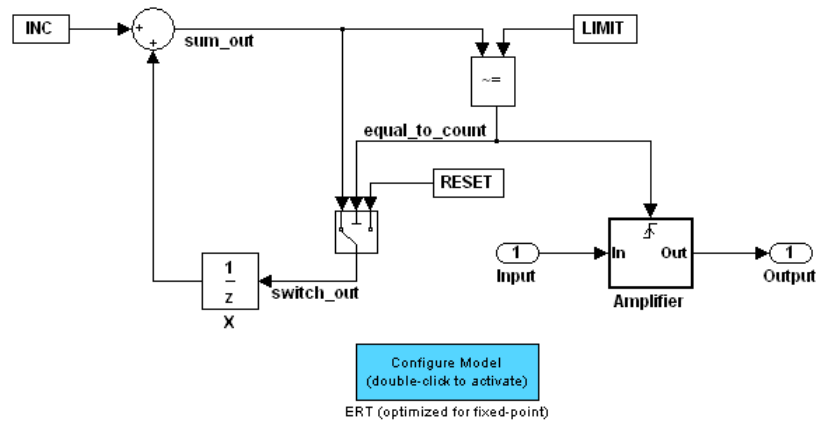
- 1 Open the model that you want to configure.
- 2 Open the Real-Time Workshop Embedded Coder library by typing the command `rtweclib`.
- 3 The top level of the library is shown below.



- 4 Double-click the Configuration Wizards icon. The Configuration Wizards sublibrary opens, as shown below.

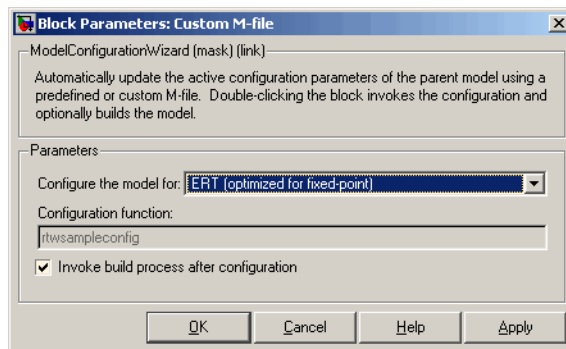


- 5 Select the Configuration Wizard block you want to use and drag and drop it into your model. In the figure below, the ERT (optimized for fixed-point) Configuration Wizard block has been added to the model.



- 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 8-64.

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 M-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:

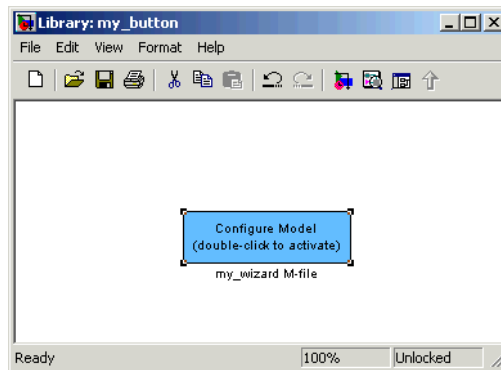
```
disp('Custom Configuration Wizard Script completed.');
```

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:

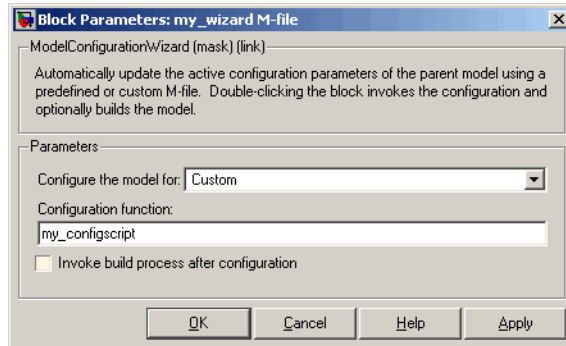
- 1 Open the Real-Time Workshop Embedded Coder library and the Configuration Wizards sublibrary, as described in “Adding a Configuration Wizard Block to Your Model” on page 8-61.
- 2 Select **New Library** from the **File** menu of the Configuration Wizards sublibrary window. An empty library window 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 M-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');
```

Replacement of `STF_rtw_info_hook` Mechanism

Prior to MATLAB Release 14, custom targets supplied target-specific information with a hook file (referred to as `STF_rtw_info_hook.m`). The `STF_rtw_info_hook` specified properties such as word sizes for integer data types (for example, `char`, `short`, `int`, and `long`), and C implementation-specific properties of the custom target.

The `STF_rtw_info_hook` mechanism has been replaced by the **Hardware Implementation** pane of the Configuration Parameters dialog box. Using this dialog box, you can specify all properties that were formerly specified in your `STF_rtw_info_hook` file.

For backward compatibility, existing `STF_rtw_info_hook` files continue to operate correctly. However, you should convert your target and models to use of the **Hardware Implementation** pane. See the “Describing Hardware Properties” section of the Real-Time Workshop documentation.

Optimizing Task Scheduling for Multirate Multitasking Models on RTOS Targets

In this section...

“Overview” on page 8-73

“Using `rtmStepTask`” on page 8-74

“Task Scheduling Code for Multirate Multitasking Model on Wind River Systems VxWorks Target” on page 8-74

“Suppressing Redundant Scheduling Calls” on page 8-75

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 sub-rate 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 sub-rate task. When a given sub-rate 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 sub-rate 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** option, 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
```


Target Function Libraries

- “Introduction to Target Function Libraries” on page 9-2
- “Creating Function Replacement Tables” on page 9-14
- “Examining and Validating Function Replacement Tables” on page 9-76
- “Registering Target Function Libraries” on page 9-84
- “Target Function Library Limitations” on page 9-92

Introduction to Target Function Libraries

In this section...

“Overview of Target Function Libraries” on page 9-2

“Target Function Libraries General Workflow” on page 9-5

“Target Function Libraries Quick-Start Example” on page 9-7

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 three 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
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

7. GNU is a registered trademark of the Free Software Foundation.

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 three 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_sinfncns.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')
- 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:

Floating-Point Math Functions			
abs	cos	log10	tan
acos	cosh	pow (Simulink)/power (Embedded MATLAB)	tanh
asin	exp	sin	

atan	floor	sinh	
ceil	log	sqrt	
Copy Utility Function			
memcpy			
Nonfinite Support Utility Functions			
getInf	getMinusInf	getNaN	

The TFL API also supports the following scalar operators for replacement with custom library functions using TFL tables:

- + (addition)
- (subtraction)
- * (multiplication)
- / (division)

Target Function Libraries General Workflow

The general steps for creating and using a target function library are as follows:

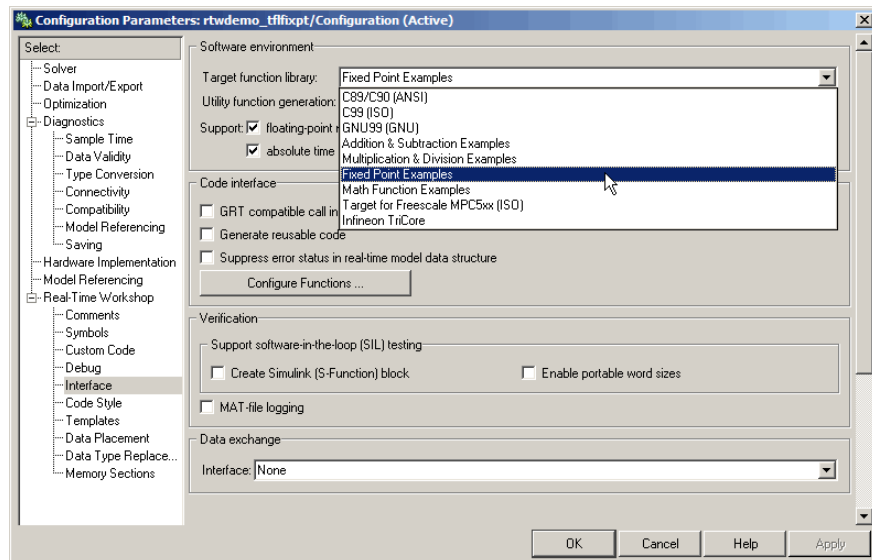
- 1 Create one or more TFL tables containing replacement entries for math operators (+, -, *, /) and functions using an API based on the MATLAB API. (The demo `rtwdemo_tfl_script` provides example tables that can be used as a starting point for customization.)

Name	Implementation	NumIn	In1Type	In2Type	RTW_OP_MUL
RTW_OP_ADD	u16_add_SameSlopeZeroBias	2	uint16	uint16	
RTW_OP_ADD	s32_add_SameSlopeZeroBias	2	int32	int32	
RTW_OP_DIV	s16_div_s16_s16_slopebias	2	fixd(1,16,15,2)	fixd(1)	
RTW_OP_DIV	s32_div_s16_s16_binarypoint	2	fixd(1,16,15)	fixd(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	fixd(1,16,15)	fixd(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 M 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, select the desired target function library from the **Target function library** drop-down list located on the **Interface** pane of the Real-Time Workshop dialog box.



- 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 `tfl_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 9-14.

```
function hTable = tfl_table_sinfcn()
%TFL_TABLE_SINFCN - Describe function entries for a Target Function Library table.

hTable = RTW.TflTable;

% Create entry for double data type sine function replacement
hTable.registerCFunctionEntry(100, 1, 'sin', 'double', 'sin_dbl', ...
                             'double', '<sin_dbl.h>', '', '');

% Create entry for single data type sine function replacement
hTable.registerCFunctionEntry(100, 1, 'sin', 'single', 'sin_sgl', ...
                             'double', '<sin_sgl.h>', '', '');
```

Note See “Example: Mapping Floating-Point Math Functions to Target-Specific Implementations” on page 9-24 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 = tfl_table_sinfcn

tbl =

RTW.TflTable
      Version: '1.0'
```

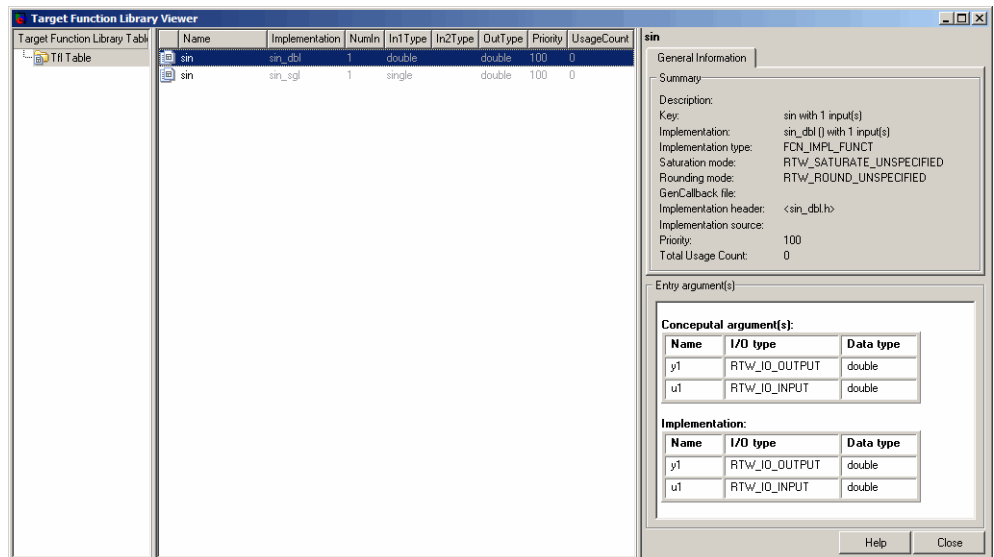
```
AllEntries: [2x1 RTW.Tf1CFunctionEntry]
ReservedSymbols: []
```

```
>>
```

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 9-76.)

- 4 Create and save a TFL registration file that includes the `tf1_table_sinfcn` table, such as the `s1_customization.m` file shown below. The file specifies that the TFL to be registered is named 'Sine Function Example' and

consists of `tfl_table_sinfcn`, with the default ANSI⁸ 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_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 `sl_customization.m` file in the MATLAB search path or in the current working directory, the TFL is registered at each Simulink startup.

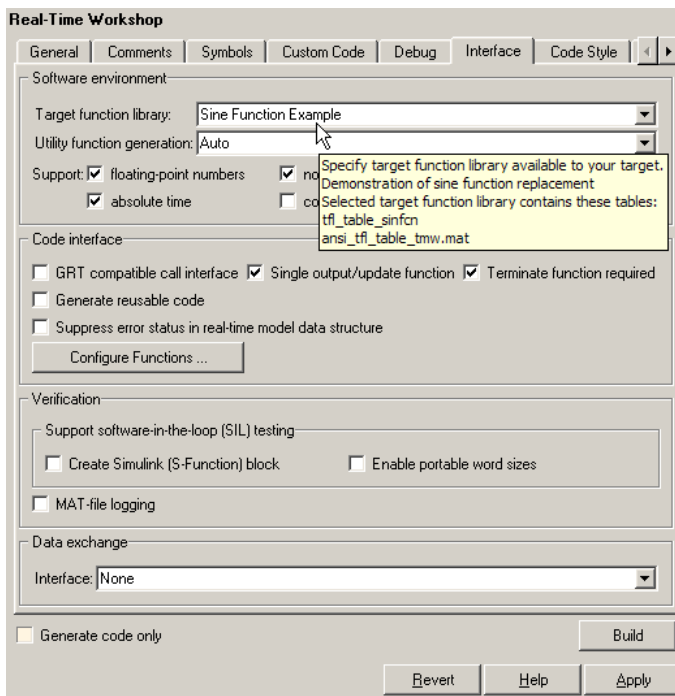
Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

8. 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 9-84.

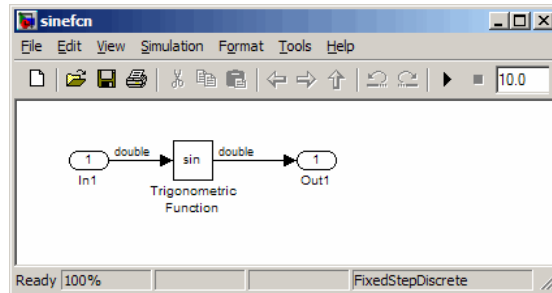
- 5 With your `sl_customization.m` file in the MATLAB search path or in the current working directory, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified 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.



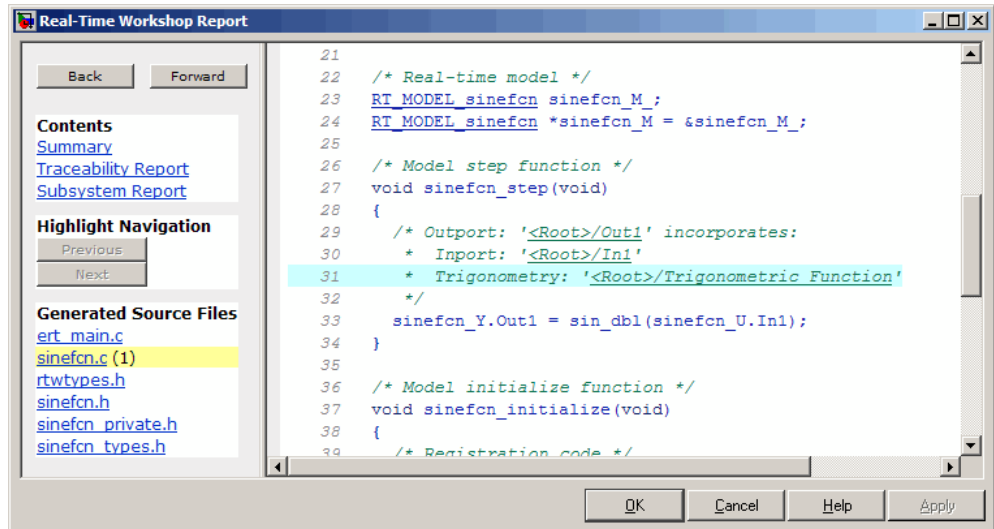
Optionally, you can relaunch the TFL Viewer, using the command `RTW.viewTFL` with no argument, to examine all registered TFLs, including Sine Function Example.

- 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.



- 9 If functions were not replaced as you intended, you can use the techniques described in “Examining and Validating Function Replacement Tables” on page 9-76 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.TflCFunctionEntry]
    MissCache: [0x1 handle]
    TLCCallList: [0x1 handle]
    TflTables: [2x1 RTW.TflTable]

>> a.HitCache(1)
```

```
ans =  
  
RTW.Tf1CFunctionEntry  
    Key: 'sin'  
    Priority: 100  
    ConceptualArgs: [2x1 RTW.Tf1ArgNumeric]  
    Implementation: [1x1 RTW.CImplementation]  
  
.  
.  
.  
>>
```

Creating Function Replacement Tables

In this section...

“Overview of Function Replacement Table Creation” on page 9-14

“Creating Table Entries” on page 9-17

“Example: Mapping Floating-Point Math Functions to Target-Specific Implementations” on page 9-24

“Example: Mapping the memcpy Function to a Target-Specific Implementation” on page 9-29

“Example: Mapping Nonfinite Support Utility Functions to Target-Specific Implementations” on page 9-34

“Example: Mapping Operators to Target-Specific Implementations” on page 9-39

“Mapping Fixed-Point Operators to Target-Specific Implementations” on page 9-45

“Specifying Build Information for Function Replacements” on page 9-70

“Adding Target Function Library Reserved Identifiers” on page 9-74

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 M-file containing a function definition in the following general form:

```
function hTable = tf1_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 9-7:

```
function hTable = tf1_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 9-17 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 9-7:

```
% 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 M-file using the name of the table definition function, for example, `tfl_table_sinfcn.m`.

After you have created a table definition M-file, you can do the following:

- Examine and validate the table, as described in “Examining and Validating Function Replacement Tables” on page 9-76.
- Register a TFL containing the table with the Simulink software, as described in “Registering Target Function Libraries” on page 9-84.

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
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
Alternative method for function entry creation	
registerCFunctionEntry	Create TFL function entry based on specified parameters and register in TFL table

Function	Description
<code>registerCPromotableMacroEntry</code>	Create TFL promotable macro entry based on specified parameters and register in TFL table (for <code>abs</code> function replacement only)
Build information	
<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
Reserved identifiers	
<code>setReservedIdentifiers</code>	Register specified reserved identifiers to be associated with TFL table

Creating Table Entries

- “Overview of Table Entry Creation” on page 9-17
- “General Method for Creating Function and Operator Entries” on page 9-19
- “Alternative Method for Creating Function Entries” on page 9-23

Overview of Table Entry Creation

You define TFL table entries by issuing TFL table creation function calls inside a table definition M-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_SINFCN - 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 9-19, 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
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
setTflCFunctionEntryParameters	Set specified parameters for function entry in TFL table

Function	Description
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 9-23.

Function	Description
Alternative method for function entry creation	
registerCFunctionEntry	Create TFL 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

- Implementation argument order must match the conceptual argument order. Remapping the argument order in your 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 *un*), you can use a simpler alternative method for creating the entries, as described in “Alternative Method for Creating Function Entries” on page 9-23.
-

1 Within the function body of your table definition M-file, instantiate a TFL table entry for a function or operator, using one of the following lines of code:

- `fcn_entry = RTW.Tf1CFunctionEntry;`
- `op_entry = RTW.Tf1COperationEntry;`
- `op_entry = RTW.Tf1COperationEntryGenerator;`

(`RTW.Tf1COperationEntryGenerator` provides advanced fixed-point parameters, described in “Mapping Fixed-Point Operators to Target-Specific Implementations” on page 9-45, that are not available in `RTW.Tf1COperationEntry`.)

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 Floating-Point Math Functions to Target-Specific Implementations” on page 9-24 and “Example: Mapping Operators to Target-Specific Implementations” on page 9-39. 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, `y1`
 - For input argument names, `u1`, `u2`, ..., `un`

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`
- `registerCPromotableMacroEntry` (use only for the `abs` function)

For example:

```
hTable = RTW.TflTable;

registerCFunctionEntry(hTable, 100, 1, 'sqrt', 'double', ...
    'sqrt', 'double', '<math.h>', '', '');
```

For detailed descriptions of the function arguments, see the `registerCFunctionEntry` and `registerCPromotableMacroEntry` reference pages in the Real-Time Workshop Embedded Coder documentation.

Example: Mapping Floating-Point Math Functions to Target-Specific Implementations

The Real-Time Workshop Embedded Coder software supports the following floating-point math functions for replacement with custom library functions using target function library (TFL) tables.

<code>abs</code>	<code>cos</code>	<code>log10</code>	<code>tan</code>
<code>acos</code>	<code>cosh</code>	<code>pow</code> (Simulink)/ <code>power</code> (Embedded MATLAB)	<code>tanh</code>
<code>asin</code>	<code>exp</code>	<code>sin</code>	
<code>atan</code>	<code>floor</code>	<code>sinh</code>	
<code>ceil</code>	<code>log</code>	<code>sqrt</code>	

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 9-19 to create a TFL table entry for the `sin` function.

Note See “Target Function Libraries Quick-Start Example” on page 9-7 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 9-23.

- 1 Create and save the following TFL table definition file, `tfl_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 sine 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(tbl)`). For more information about validating TFL tables, see “Examining and Validating Function Replacement Tables” on page 9-76.
- 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 `tfl_table_sinfcn2`, 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

% 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 directory, so that the TFL is registered at each Simulink startup.

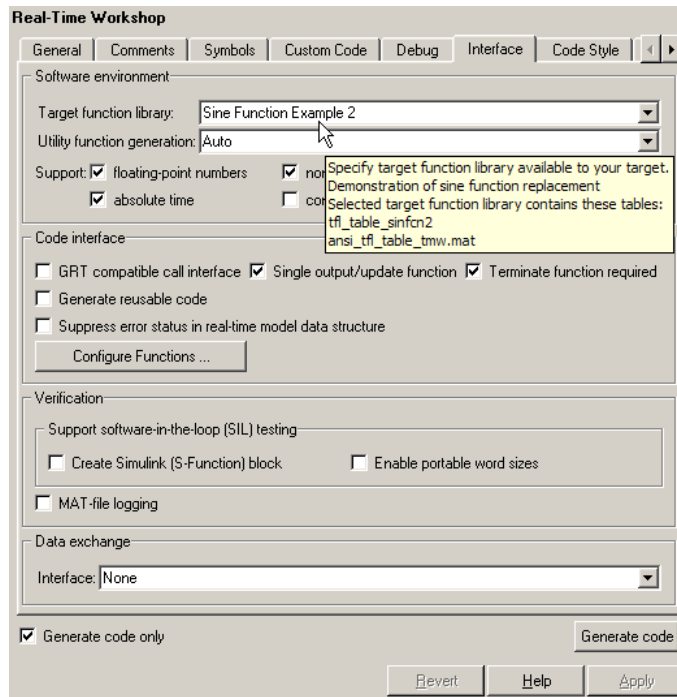
Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

9. 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 9-84.

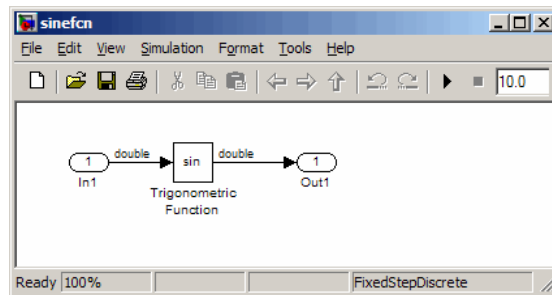
- 4 With your `sl_customization.m` file in the MATLAB search path or in the current working directory, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified 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.



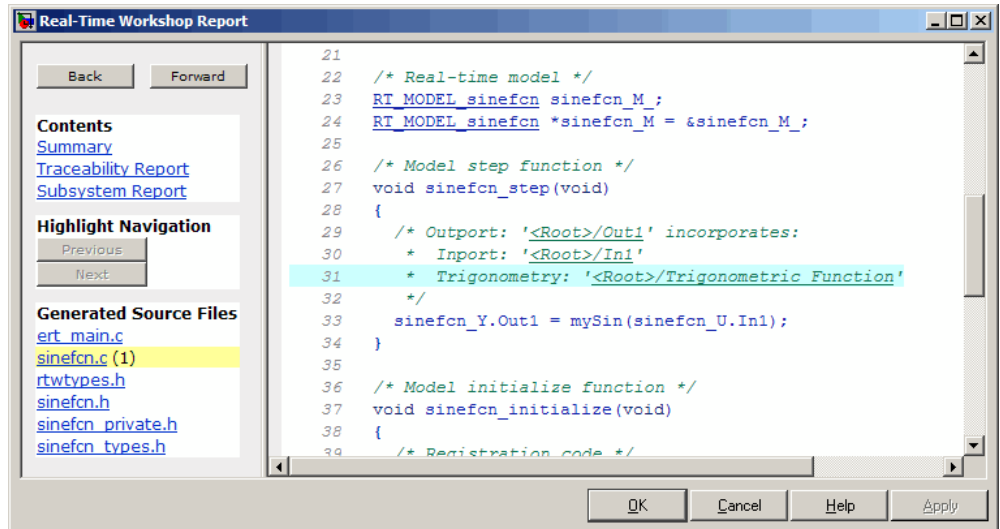
Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including Sine Function Example 2.

- 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.



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 9-19 to create a TFL table entry for the `memcpy` function.

- 1 Create and save the following TFL table definition file, `tf1_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 = tf1_table_memcpy()
%TFL_TABLE_MEMCPY - Describe memcpy function entry for a TFL table.
```

```

hTable = RTW.Tf1Table;

% 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 9-76.
- 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 ANSI¹⁰ 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 directory, so that the TFL is registered at each Simulink startup.

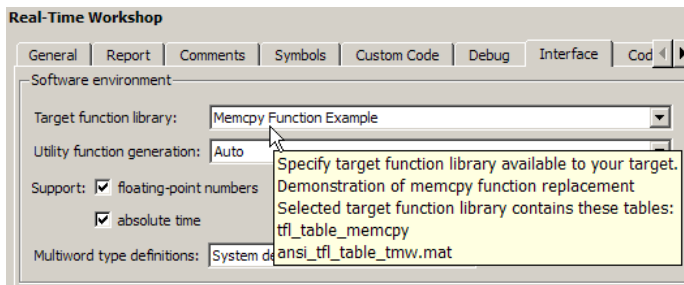
Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh Embedded MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

10. 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 9-84.

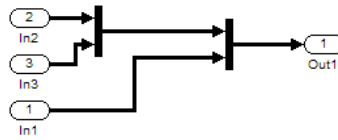
- 4 With your `sl_customization.m` file in the MATLAB search path or in the current working directory, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified 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.

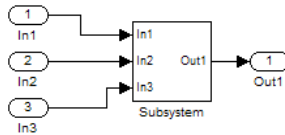


Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including Memcpy Function Example.

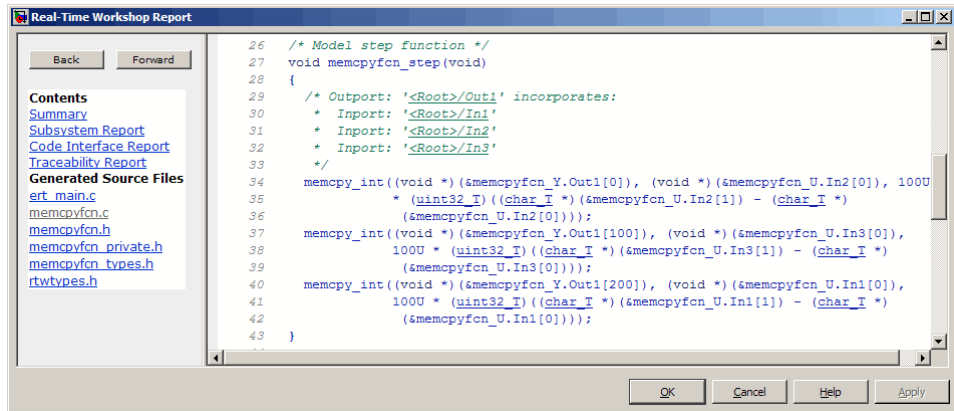
- 5 Create an ERT-based model that uses `memcpy` for vector assignments. For example,
 - a Use In, Out, and Mux blocks to create the following model. (Alternatively, you can open `rtwdemo_tflmath/Subsystem1` and copy the subsystem contents to a new model.)



- 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.



The screenshot shows a window titled "Real-Time Workshop Report". On the left is a navigation pane with buttons for "Back" and "Forward", and a "Contents" section listing various reports and source files. The main area displays C code for a function named `memcpyfcn_step`. The code includes comments about model step functions and incorporates output and input ports. It uses `memcpy_int` to copy data from input ports to output ports.

```

26  /* Model step function */
27  void memcpyfcn_step(void)
28  {
29  /* Output: '<Root>/Out1' incorporates:
30  * Input: '<Root>/In1'
31  * Input: '<Root>/In2'
32  * Input: '<Root>/In3'
33  */
34  memcpy_int((void *)(&memcpyfcn_Y.Out1[0]), (void *)(&memcpyfcn_U.In2[0]), 100U
35            * (uint32_T)((char_T *)(&memcpyfcn_U.In2[1]) - (char_T *)
36              (&memcpyfcn_U.In2[0]));
37  memcpy_int((void *)(&memcpyfcn_Y.Out1[100]), (void *)(&memcpyfcn_U.In3[0]),
38            100U * (uint32_T)((char_T *)(&memcpyfcn_U.In3[1]) - (char_T *)
39              (&memcpyfcn_U.In3[0]));
40  memcpy_int((void *)(&memcpyfcn_Y.Out1[200]), (void *)(&memcpyfcn_U.In1[0]),
41            100U * (uint32_T)((char_T *)(&memcpyfcn_U.In1[1]) - (char_T *)
42              (&memcpyfcn_U.In1[0]));
43  }

```

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 9-19 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 9-76.
- 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 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_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.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

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

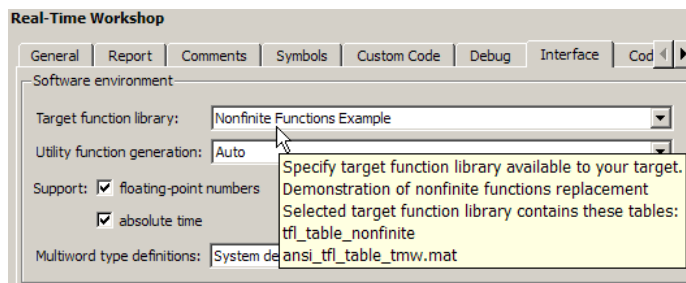
Place this `sl_customization.m` file in the MATLAB search path or in the current working directory, so that the TFL is registered at each Simulink startup.

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 9-84.

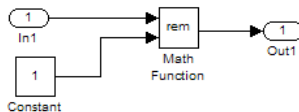
- 4 With your `sl_customization.m` file in the MATLAB search path or in the current working directory, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified 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 and configure the **Signal Attributes** for the `In1` and `Constant` source blocks. For each, 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.

The screenshot shows a window titled "Real-Time Workshop Report". On the left is a "Contents" sidebar with links to various reports and source files. The main area displays C code for a function named `rt_InitInfAndNaN`. The code includes headers, defines `NumBitsPerChar` as 8, and declares several real data types. The function itself initializes `rtNaN`, `rtNaNF`, `rtInf`, `rtInfF`, `rtMinusInf`, and `rtMinusInfF` by calling their respective `get` functions.

```

13  /*
14  * Abstract:
15  *   Real-Time Workshop function to initialize non-finites,
16  *   (Inf, NaN and -Inf).
17  */
18  #include "rt_nonfinite.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 rtInfF;
27  real32_T rtMinusInfF;
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      rtInfF = getInfF();
42      rtMinusInf = getMinusInf();
43      rtMinusInfF = getMinusInfF();
44  }

```

Example: Mapping 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:

- + (addition)
- (subtraction)
- * (multiplication)
- / (division)

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 9-19 to create a TFL table entry for the + (addition) operator.

- 1 Create and save the following TFL table definition file, `tfl_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 = 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
% Saturation on, Rounding no preference
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(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 = 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 );

copyConceptualArgsToImplementation(op_entry);

addEntry(hTable, op_entry);
```


- 2** Optionally, perform a quick check of the validity of the sine function entry by invoking the table definition file at the MATLAB command line (`>> tbl = tfl_table_add_uint8`) and by viewing it in the TFL Viewer (`>> RTW.viewTfl(tfl_table_add_uint8)`).

For more information about validating TFL tables, see “Examining and Validating Function Replacement Tables” on page 9-76.

- 3** Create and save the following TFL registration file, which references the `tfl_table_add_uint8` table.

The file specifies that the TFL to be registered is named 'Addition Operator Example' and consists of `tfl_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.TargetHWDDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

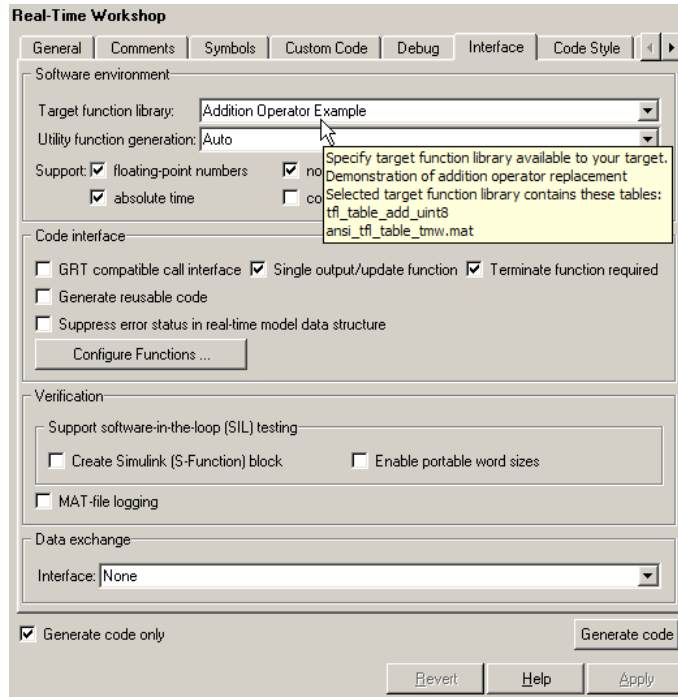
Place this `sl_customization.m` file in the MATLAB search path or in the current working directory, so that the TFL is registered at each Simulink startup.

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 9-84.

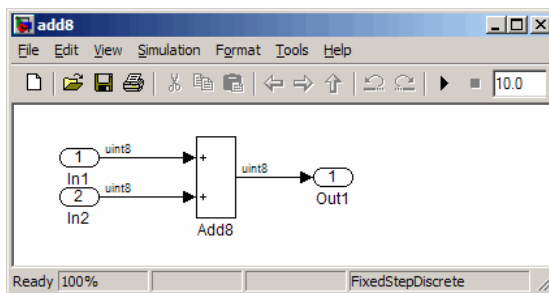
- 4 With your `sl_customization.m` file in the MATLAB search path or in the current working directory, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified 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.



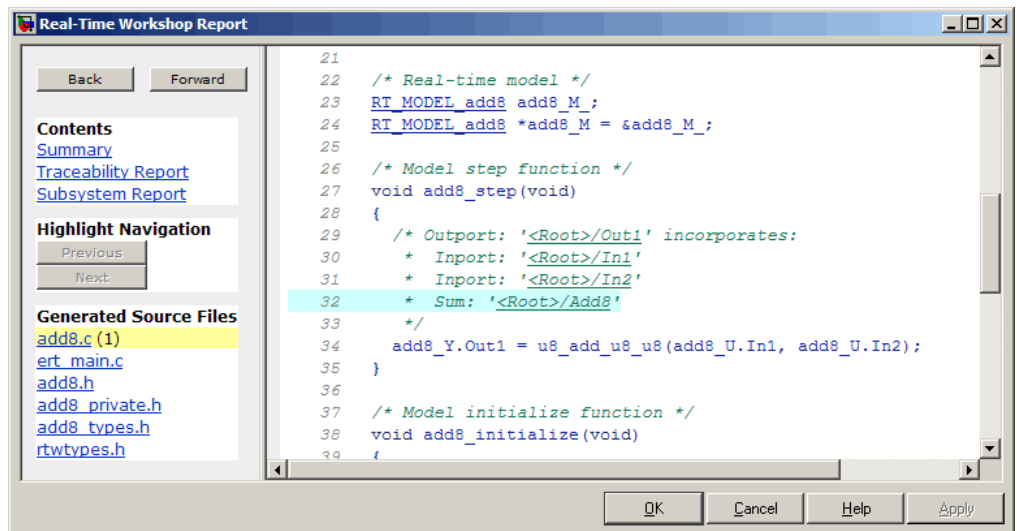
Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including Addition Operator Example.

- 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 Fixed-Point Operators to Target-Specific Implementations

- “Overview of Fixed-Point Operator Replacement” on page 9-45
- “Fixed-Point Numbers and Arithmetic” on page 9-46
- “Creating Fixed-Point Operator Entries” on page 9-51
- “Example: Creating Fixed-Point Operator Entries for Binary-Point-Only Scaling” on page 9-54
- “Example: Creating Fixed-Point Operator Entries for [Slope Bias] Scaling” on page 9-58
- “Example: Creating Fixed-Point Operator Entries for Relative Scaling (Multiplication and Division)” on page 9-61
- “Example: Creating Fixed-Point Operator Entries for Net Slope (Multiplication and Division)” on page 9-64
- “Example: Creating Fixed-Point Operator Entries for Equal Slope and Zero Net Bias (Addition and Subtraction)” on page 9-67

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:

- + (addition)
- (subtraction)
- * (multiplication)
- / (division)

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.

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:

$$(S_o Q_o + B_o) = (S_1 Q_1 + B_1) < op > (S_2 Q_2 + B_2)$$

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 $V0 = V1 + V2$ implies that

$$Q_o = \left(\frac{S_1}{S_o} \right) Q_1 + \left(\frac{S_2}{S_o} \right) Q_2 + \left(\frac{B_1 + B_2 - B_o}{S_o} \right)$$

If an addition replacement function is defined such that the scaling on the operands and sum are equal and the net bias

$$\left(\frac{B_1 + B_2 - B_o}{S_o} \right)$$

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 `setTf1COperationEntryParameters`.)

Subtraction. The operation $V0 = V1 - V2$ implies that

$$Q_o = \left(\frac{S_1}{S_o} \right) Q_1 - \left(\frac{S_2}{S_o} \right) Q_2 + \left(\frac{B_1 - B_2 - B_o}{S_o} \right)$$

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 `setTflCOperationEntryParameters`.) 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 `NetSlopeFixedExponent`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Note When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetSlopeFixedExponent`, 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 `NetSlopeFixedExponent`. (For parameter descriptions, see the reference page for the function `setTf1COperationEntryParameters`.)

Note When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetSlopeFixedExponent`, matching entries must have arguments with zero bias.

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 9-19 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

Function	Description
createAndSetCImplementationReturn	Create implementation return argument from specified properties and add to implementation for TFL table entry
setTf1COperationEntryParameters	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 9-54.</p>	RTW.Tf1COperationEntry	<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).

To match...	Instantiate class...	Minimally specify parameters...
<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 9-58.</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).
<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 9-61.</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'.

To match...	Instantiate class...	Minimally specify parameters...
<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 9-64.</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). • NetSlopeFixedExponent: 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'.
<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 9-67.</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 9-19 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);
```



```

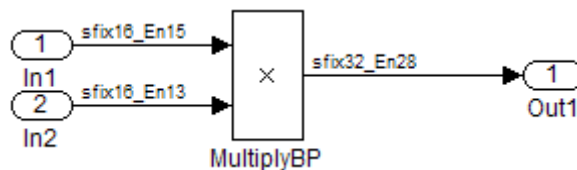
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
                               'Name',      'u1', ...
                               'IOType',    'RTW_IO_INPUT', ...
                               'IsSigned',  true, ...
                               'WordLength', 16, ...
                               '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 Operators to Target-Specific Implementations” on page 9-39, 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 9-19 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);

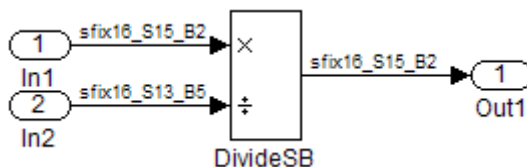
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
                               'Name',      'u1', ...
                               'IOType',    'RTW_IO_INPUT', ...
                               'IsSigned',   true, ...
                               'WordLength', 16, ...
                               '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 Operators to Target-Specific Implementations” on page 9-39, 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 **Round integer calculations toward** 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 9-19 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_rsfOp125`). 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);

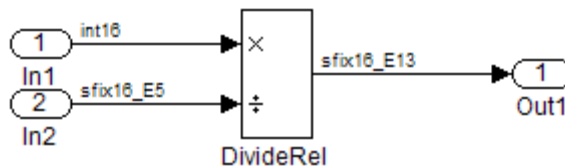
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
        'Name',          'u1', ...
        'IOType',        'RTW_IO_INPUT', ...
        'IsSigned',      true, ...
        'WordLength',    16, ...
        'FractionLength', 0);

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 Operators to Target-Specific Implementations” on page 9-39, 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 **Round integer calculations toward** 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 9-19 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 `NetSlopeFixedExponent`.
- 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 `NetSlopeFixedExponent` 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).

```
function hTable = make_net_slope_tfl_table()
%MAKE_NET_SLOPE_TFL_TABLE - Describe entries for a Target Function Library table.
hTable = RTW.TflTable;

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,dtv,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(dtv),...
    typeStrFunc(dtnum),...
    typeStrFunc(dtden));

op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
setTflCOperationEntryParameters(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',      dtypes.Signed,...
    'WordLength',    dtypes.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',      dtypes.Signed,...
    'WordLength',    dtypes.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',      dtypes.Signed,...
    'WordLength',    dtypes.WordLength,...
    'Bias',          0);

arg = getTf1ArgFromString(hTable, 'y1', typeStrBase(dtypes));
op_entry.Implementation.setReturn(arg);

arg = getTf1ArgFromString(hTable, 'u1', typeStrBase(dtypes));
op_entry.Implementation.addArgument(arg);

```

```

arg = getTflArgFromString(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 9-19 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.Tf1COperationEntryGenerator` class, which provides access to the fixed-point parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias`.
- The function `setTf1COperationEntryParameters` 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);

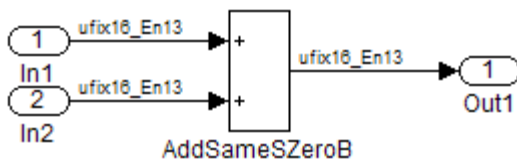
createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      false, ...
    'WordLength',    16, ...
    'FractionLength', 0);

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 Operators to Target-Specific Implementations” on page 9-39, 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 **Round integer calculations toward** to `Zero`

Specifying Build Information for Function Replacements

- “Functions for Specifying Table Entry Build Information” on page 9-71
- “Using `RTW.copyFileToBuildDir` to Copy Files to the Build Directory” on page 9-72
- “`RTW.copyFileToBuildDir` Examples” on page 9-72

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` 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 Directory

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 directory 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 directory. 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 directory 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`
- `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 directory and must be copied into the build directory 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_mu1.h` contains an inlined function that invokes assembly functions contained in `s32_mu1.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 directory.


```

function hTable = make_my_tfl_table

hTable = RTW.TflTable;

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.TflTable;

% 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');
addAdditionalLinkObjsPath(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 the base list of TFL reserved identifiers, see “Reserved 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, ISO¹², or GNU¹³ 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 information and examples, see `setReservedIdentifiers` in the Real-Time Workshop Embedded Coder reference documentation.

12. ISO is a registered trademark of the International Organization for Standardization.

13. GNU is a registered trademark of the Free Software Foundation.

Examining and Validating Function Replacement Tables

In this section...

“Overview of Function Replacement Table Validation” on page 9-76

“Invoking the Table Definition M-File” on page 9-76

“Using the Target Function Library Viewer to Examine Your Table” on page 9-77

“Using the Target Function Library Viewer to Examine Registered TFLs” on page 9-78

“Tracing Code Generated Using Your Target Function Library” on page 9-80

“Examining TFL Cache Hits and Misses” on page 9-82

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 M-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 M-File

Immediately after creating or modifying a table definition M-file (as described in “Creating Function Replacement Tables” on page 9-14), 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 = tfl_table_sinfcn
```

```
tbl =
```

```

RTW.Tf1Table
    Version: '1.0'
    AllEntries: [2x1 RTW.Tf1CFunctionEntry]
    ReservedSymbols: []

>>

```

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', ...

>>

```

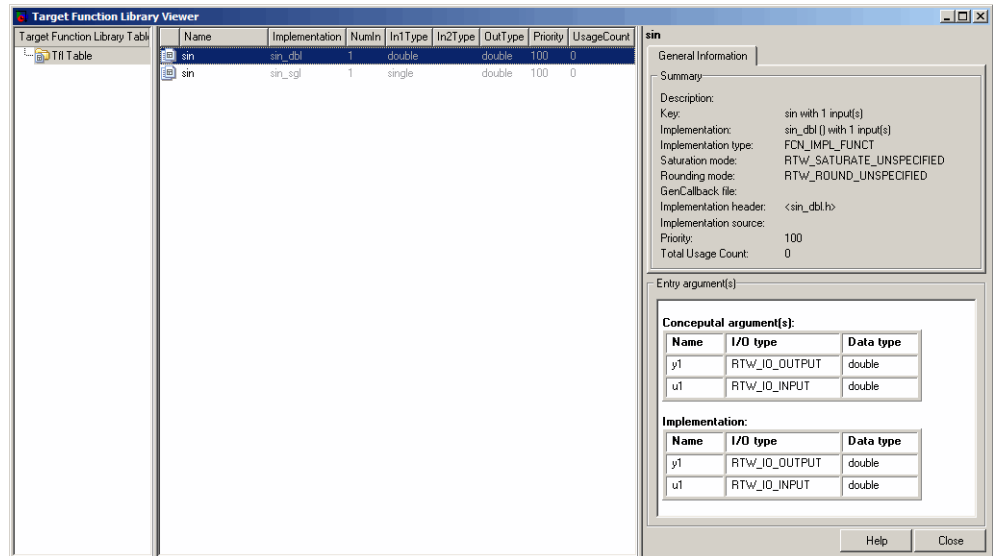
Using the Target Function Library Viewer to Examine Your Table

After creating or modifying a table definition M-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).

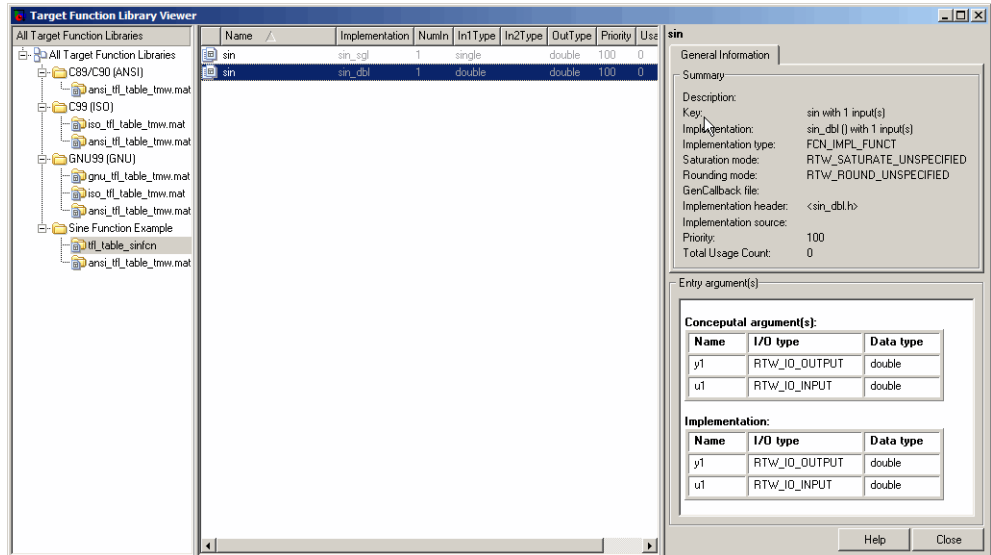
For more information about the TFL Viewer, see “Using the Target Function Library Viewer” in the Real-Time Workshop documentation.

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 9-84), 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, or you may need to issue the MATLAB command `sl_refresh_customizations`.

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

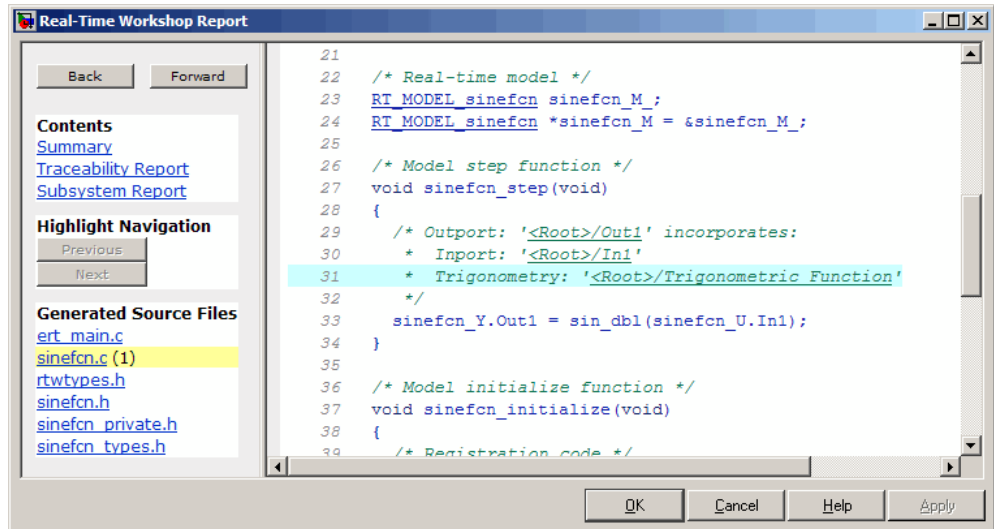
- Incorrect relative order of tables in the library (tables are displayed in search order)
- Table entry problems as listed in the previous section

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 the HTML report, as shown in the following example.



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 9-77 and “Using the Target Function Library Viewer to Examine Registered TFLs” on page 9-78
- 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 9-82

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 9-80.

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.

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 9-84

“Using the `sl_customization` API to Register a TFL with Simulink Software” on page 9-85

“Using the `rtwTargetInfo` API to Register a TFL with Embedded MATLAB Coder Software” on page 9-89

“Registering Multiple TFLs” on page 9-90

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 M-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 M-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 an object called the `Simulink.CustomizationManager`. The function is declared as follows:

```
function sl_customization(cm)
```

The body of the `sl_customization` function invokes the `registerTargetInfo(tfl)` method provided by `Simulink.CustomizationManager` 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 `tfl = 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	Cell array of strings specifying the tables that make up the TFL, in descending priority order.
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)', or an equivalent alias.</p> <hr/>
TargetHWDeviceType	Always specify {'*'}

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

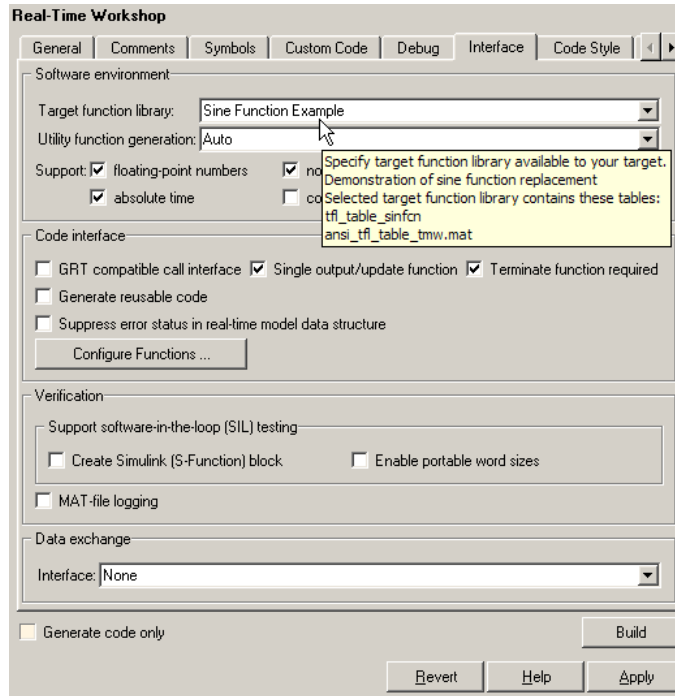
% Local function to define a TFL containing tfl_table_sinfcn
function thisTfl = locTflRegFcn

% 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 `sl_customization.m` file containing this function in the MATLAB search path or in the current working directory, 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(tf1)` 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 9-85.

```
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 directory, 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 excerpt illustrates the general approach, which applies equally to Simulink and Embedded MATLAB Coder TFL registration files.

```
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 operator 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_tflmath.mdl
thisTfl(4) = RTW.TflRegistry;
thisTfl(4).Name = 'Math Function Examples';
thisTfl(4).Description = 'Demonstration of math function replacement';
thisTfl(4).TableList = {'tfl_table_math'};
thisTfl(4).BaseTfl = 'C89/C90 (ANSI)';
thisTfl(4).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, you can inspect the generated code.

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 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 9-76.

ERT Target Requirements, Restrictions, and Control Files

- “Requirements and Restrictions for ERT-Based Simulink Models” on page 10-2
- “ERT System Target File and Template Makefiles” on page 10-4

Requirements and Restrictions for ERT-Based Simulink Models

- For code generation with Real-Time Workshop Embedded Coder software, configure your model for the following options on the **Solver** pane of the Configuration Parameters dialog box:
 - **Type:** `fixed-step`
 - **Solver:** You can select any available solver algorithm.
 - **Tasking mode for periodic sample times:** When the model is single-rate, you must select the `SingleTasking` or `Auto` mode. Permitted Solver Modes for Real-Time Workshop® Embedded Coder™ Targeted Models on page 2-14 indicates permitted solver modes for single-rate and multirate models.
- If you use blocks that have a dependency on absolute time in a program, you should properly specify the **Application lifespan (days)** parameter on the **Optimization** pane. (See “Limitations on the Use of Absolute Time” in the Real-Time Workshop documentation for a list of such blocks.) You can use these blocks in applications that run for extremely long periods, with counters that provide accurate and overflow-free absolute time values, provided that you specify a long enough lifespan. If you are designing a program that is intended to run indefinitely, specify **Application lifespan (days)** as `inf`. This generates a 64 bit integer counter. For an application whose sample rate is 1000 MHz, a 64 bit counter will not overflow for more than 500 years.
- You can use any Simulink blocks in your models, except for blocks not supported by the Embedded-C format, as follows:
 - MATLAB Fcn block
 - M-file and Fortran S-functions that are not inlined with TLC

Note that use of certain blocks is not recommended for production code generation for embedded systems. To view a table that summarizes characteristics of blocks in the Simulink block library, execute the following command at the MATLAB command line:

```
showblockdatatypetable
```

Refer to the **Code Generation Support** column of the table and its footnotes, including the footnote “Not recommended for production code.”

- You can use both inlined and non-inlined S-functions with the Real-Time Workshop Embedded Coder product. However, inlined S-functions are often advantageous in production code generation, for example in implementing device drivers. See “Tradeoffs in Device Driver Development” in the Developing Embedded Targets document for a discussion of the pros and cons.

ERT System Target File and Template Makefiles

The Real-Time Workshop Embedded Coder system target file is `ert.tlc`.

The Real-Time Workshop product provides template makefiles for the Real-Time Workshop Embedded Coder software in the following development environments:

- `ert_lcc.tmf` — LCC compiler
- `ert_tornado.tmf` — Wind River Systems Tornado® (VxWorks)
- `ert_unix.tmf` — The Open Group UNIX host
- `ert_vc.tmf` — Microsoft Visual C++®
- `ert_msvc.tmf` — Visual C++, project file only
- `ert_watc.tmf` — Watcom C

Examples

Use this list to find examples in the documentation.

Data Structures, Code Modules, and Program Execution

- “Real-Time Model (rtModel) Data Structure” on page 2-2
- “rtModel Accessor Macros” on page 2-3
- “Rate Grouping and Rate-Specific model_step Functions” on page 2-17
- “Rate Grouping and the Static Main Program” on page 2-27
- “Making Your S-Functions Rate Grouping Compliant” on page 2-31

Code Generation

- “Using Virtualized Output Ports Optimization” on page 3-38
- “Creating and Using a Code Generation Report” on page 3-43
- “Generating an ERT S-Function Wrapper” on page 3-69
- “Techniques for Exporting Function-Call Subsystems” on page 3-75
- “Function-Call Subsystem Export Example” on page 3-77
- “Examples of Modular Function Code for Nonvirtual Subsystems” on page 3-88
- “Model Function Prototypes Example” on page 3-105
- “Sample M Script for Configuring Model Function Prototypes” on page 3-114
- “C++ Encapsulation Quick-Start Example” on page 3-119
- “Sample M-Script for Configuring the Step Method for a Model Class” on page 3-136
- “Generating a Shared Library Version of Your Model Code” on page 3-141
- “Creating Application Code to Load and Use Your Shared Library File” on page 3-142
- “Importing an AUTOSAR Software Component” on page 5-8

Verifying Generated Code

- “Verifying Generated Code with Software-in-the-Loop” on page 4-2
- “Using PIL Mode in the Model Block” on page 4-22
- “Demos of the Target Connectivity API” on page 4-29

Custom Storage Classes

- “Setting Custom Storage Class Properties” on page 6-9
- “Generating Code with CSCs” on page 6-11
- “Creating Packages with CSC Definitions” on page 6-16
- “Defining Advanced Custom Storage Class Types” on page 6-42
- “Example of Generated Code with GetSet Custom Storage Class” on page 6-48
- “Assigning a Custom Storage Class to Data” on page 6-59

Memory Sections

- “Requirements for Defining Memory Sections” on page 7-4
- “Defining Memory Sections” on page 7-6
- “Applying Memory Sections” on page 7-10
- “Examples of Generated Code with Memory Sections” on page 7-18

Advanced Code Generation

- “Specifying Type Definition Location for User-Defined Data Types” on page 8-5
- “Example Build Process Customization Using `sl_customization.m`” on page 8-17
- “Using `set_param` to Set Model Parameters” on page 8-19
- “Automatic Model Configuration Using `ert_make_rtw_hook`” on page 8-20
- “Using the Optimized ERT Targets” on page 8-28
- “Generating Source and Header Files with a Custom File Processing (CFP) Template” on page 8-40
- “Creating a Custom File Banner Template” on page 8-55
- “Customizing a Code Generation Template (CGT) File for Custom Banner Generation” on page 8-56
- “Adding a Configuration Wizard Block to Your Model” on page 8-61
- “Creating a Custom Configuration Wizard Block” on page 8-64
- “Task Scheduling Code for Multirate Multitasking Model on Wind River Systems VxWorks Target” on page 8-74

Target Function Libraries

- “Target Function Libraries Quick-Start Example” on page 9-7
- “General Method for Creating Function and Operator Entries” on page 9-19
- “Alternative Method for Creating Function Entries” on page 9-23
- “Example: Mapping Floating-Point Math Functions to Target-Specific Implementations” on page 9-24
- “Example: Mapping the memcpy Function to a Target-Specific Implementation” on page 9-29
- “Example: Mapping Nonfinite Support Utility Functions to Target-Specific Implementations” on page 9-34
- “Example: Mapping Operators to Target-Specific Implementations” on page 9-39
- “Example: Creating Fixed-Point Operator Entries for Binary-Point-Only Scaling” on page 9-54
- “Example: Creating Fixed-Point Operator Entries for [Slope Bias] Scaling” on page 9-58
- “Example: Creating Fixed-Point Operator Entries for Relative Scaling (Multiplication and Division)” on page 9-61
- “Example: Creating Fixed-Point Operator Entries for Net Slope (Multiplication and Division)” on page 9-64
- “Example: Creating Fixed-Point Operator Entries for Equal Slope and Zero Net Bias (Addition and Subtraction)” on page 9-67
- “RTW.copyFileToBuildDir Examples” on page 9-72
- “Adding Target Function Library Reserved Identifiers” on page 9-74
- “Examining and Validating Function Replacement Tables” on page 9-76
- “Using the sl_customization API to Register a TFL with Simulink Software” on page 9-85
- “Registering Multiple TFLs” on page 9-90

A

- absolute time 3-22
- auto-configuring targets
 - purpose of 8-19
- AUTOSAR 5-2
 - standards, complying with 5-5

C

- code generation options
 - Application lifespan (days) 3-32
 - code style pane 3-26
 - Configure Model Functions 3-24
 - Create code generation report 3-9
 - Custom comments 3-10
 - External mode 3-41
 - File customization template 3-27
 - Fixed-point exception protection 3-32
 - Generate an example main program 3-27
 - Generate reusable code 3-23
 - Generate scalar inlined parameters 3-12
 - GRT compatible call interface 3-23
 - Identifier format control 3-12
 - MAT-file logging
 - clearing 3-37
 - Maximum identifier length 3-12
 - Minimum mangle length 3-12
 - Parameter structure 3-31
 - Pass root-level I/O as 3-24
 - Requirements in block comments 3-10
 - Reusable code error diagnostic 3-25
 - Simulink block descriptions 3-10
 - Simulink data object descriptions 3-10
 - Single output/update function 3-23
 - clearing 2-30
 - Stateflow object descriptions 3-10
 - Support absolute time 3-22
 - Support complex numbers 3-22
 - Support continuous time 3-22
 - for using continuous time blocks 3-4

- limitations 3-71
- Support floating-point numbers 3-21
- Support non-finite numbers 3-22
- Support non-inlined S-functions 3-22
- Suppress error status in real-time model
 - data structure 3-24
 - Terminate function required 3-23
- code generation report 3-43
- code modules, generated 2-4
- code style
 - controlling 3-26
- code templates
 - example of use 8-40
 - generating code with 8-41
 - structure of 8-39
 - summary of API 8-50
- code, user-written 2-7
- Configuration Parameters dialog box 3-3
- Configuration Wizard buttons 8-59
- custom code generation
 - of file banners 8-53
 - with code templates 8-38
- custom file processing (CFP) template 3-27
- custom storage classes
 - assigning to data 6-59
 - code generation with 6-59
 - instance-specific attributes 6-57

D

- data initialization
 - of floats and doubles 3-31
 - of internal states 3-31
 - of root-level I/O ports 3-31
- data structures
 - real-time model 2-2
- data templates 3-27
- dialog boxes
 - AUTOSAR Options 5-10
 - Model Interface 3-97

- DO-178B 5-2
 - standards, complying with 5-28
- DO-178B Standard 5-28
- DO178-B 5-2

E

- elapsed time 3-22
- entry points, model 2-24
- ERT target
 - optimized for fixed-point 8-23
 - optimized for floating-point 8-23
- ert_main.c 2-26
- ert_main.cpp 2-26
- External mode support 3-41

F

- file banners, generation of 8-53
- file packaging 2-4

G

- generated code
 - modules 2-4
- GetSet custom storage class 6-46
- guidelines
 - MISRA-C 5-5

H

- Hardware Implementation parameters
 - configuration of 3-34
- hook files
 - STF_make_rtw_hook
 - auto-configuring models with 8-20
 - customizing build process with 8-7
- HTML code generation report 3-43

I

- identifier format control parameters 3-15
- identifier format control tokens 3-14
- IEC 61508 5-2
 - standards, complying with 5-25
- IEC 61508 Standard 5-25
- industry standards and guidelines
 - modeling and coding 5-2
- integer-only code 3-37
- integer-only code generation 3-37
- interrupts, servicing 2-11

M

- main program (ert_main)
 - generated 2-8
 - modifying 2-13
 - operation of 2-12
 - static module 2-26
 - VxWorks example 2-21
- MathWorks Automotive Advisory Board (MAAB)
 - guidelines, complying with 5-4
- MathWorks™ Automotive Advisory Board (MAAB) 5-2
- MISRA C 5-2
- MISRA-C
 - guidelines, complying with 5-5
- MISRA-C guidelines 5-5
- model entry points 2-24
- modifying rt_OneStep 2-19

N

- name mangling 3-16

P

- PIL
 - introduction 4-7

- processor-in-the-loop (PIL)
 - communications API 4-27
 - configuring 4-22
 - connectivity API demos 4-29
 - connectivity configuration 4-26
 - custom target 4-26
 - default connectivity configuration 4-22
 - host-based 4-22
 - how PIL works 4-9
 - introduction 4-7
 - limitations 4-31
 - Link products 4-12
 - Model block simulation mode 4-10
 - Model block simulation mode demo 4-22
 - model reference hierarchy 4-11
 - rtiostream API 4-27
 - target connectivity API 4-26
- program execution
 - main program 2-12
 - rt_OneStep 2-14
 - multirate multitasking operation 2-16
 - multirate singletasking operation 2-18
 - reentrancy 2-18
 - single-rate singletasking operation 2-15
- pure integer code 3-37
 - and external mode 3-42

R

- rate grouping 2-17
- real-time model data structure 2-2

- reentrant code 3-23
- requirements for Real-Time Workshop Embedded
 - Coder programs 10-2
- restrictions on Real-Time Workshop Embedded
 - Coder programs 10-2
- reusable code 3-23
- RTW.copyFileToBuildDir 9-72

S

- S-function wrapper generation 3-68
- solver modes, permitted 2-14
- source code files, generated 2-4
- stack space allocation 3-39
- standards
 - DO-178B 5-28
 - IEC 61508 5-25
- standards and guidelines, modeling and coding 5-2
- STF_make_rtw_hook function
 - arguments to 8-8
- system target file (STF)
 - ERT 10-4

T

- target connectivity API 4-26
- target function library. *See* TFL
- task identifier (tid) 2-16
- template makefiles 10-4

TFL 9-1

- build information for function
 - replacements 9-70
- cache hits and misses 9-82
- conceptual view of function or operator 9-3
- defining 9-14
- examining 9-76
- registering 9-84
- reserved identifiers 9-74
- RTW.copyFileToBuildDir 9-72
- rtwTargetInfo.m file 9-84
- selecting in MATLAB Embedded Coder 9-6
- selecting in Simulink 9-6
- sl_customization.m file 9-84
- table definition M-file 9-14
- table entry 9-3
- target-specific implementation of function or operator 9-3
- TargetFcnLibHandle 9-82
- tracing generated code 9-80
- validating 9-76
- Viewer 9-77
- workflow 9-5

tid 2-16

timer interrupts 2-11

V

virtualized output port optimization 3-38

VxWorks deployment example 2-21