

Real-Time Workshop[®] Embedded Coder

For Use with Real-Time Workshop[®]

- Modeling
- Simulation
- Implementation

User's Guide

Version 4



How to Contact The MathWorks:



www.mathworks.com
comp.soft-sys.matlab

Web
Newsgroup



support@mathworks.com
suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Technical Support
Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
Sales, pricing, and general information



508-647-7000

Phone



508-647-7001

Fax



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

Mail

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

Real-Time Workshop Embedded Coder User's Guide

© COPYRIGHT 2002–2005 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

July 2002	Online only	Version 3.0 (Release 13)
December 2003	Online only	Revised for Version 3.2 (Release 13SP1+)
June 2004	Online only	Revised for Version 4.0 (Release 14)
October 2004	Online only	Revised for Version 4.1 (Release 14SP1)
March 2005	Online only	Revised for Version 4.2 (Release 14SP2)
September 2005	Online only	Revised for Version 4.3 (Release 14SP3)

Getting Started

1

What Is Real-Time Workshop Embedded Coder?	1-2
Real-Time Workshop Embedded Coder Feature Summary	1-3
What You Need to Know to Use This Product	1-5
Prerequisites	1-5
Real-Time Workshop Embedded Coder Documentation Collection	1-5
Related Documentation	1-6
Installing Real-Time Workshop Embedded Coder	1-7
Real-Time Workshop Embedded Coder Demos	1-8

Data Structures and Program Execution

2

Data Structures and Code Modules	2-2
Real-Time Model Data Structure	2-2
Code Modules	2-4
Generating the Main Program	2-8
Program Execution	2-10
Stand-Alone Program Execution	2-11
Main Program	2-12
rt_OneStep	2-13

VxWorks Example Main Program Execution	2-20
Overview	2-20
Task Management	2-20
Model Entry Points	2-22
model_step	2-22
model_initialize	2-24
model_terminate	2-24
model_SetEventsForThisBaseStep	2-25
The Static Main Program Module	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
Listing 1: Outputs Code Generation Without Rate Grouping	2-32
Listing 2: Outputs Code Generation With Rate Grouping	2-34

Code Generation Options and Optimizations

3

Accessing the ERT Target Options	3-3
Viewing ERT Target Options in the Configuration Parameters Dialog	3-4
Support for Continuous Time Blocks and Solvers	3-5
Continuous Block Support	3-5
Continuous Solver Support	3-5
Mapping Application Requirements to Configuration Options	3-6
A Guide to the ERT Target Options	3-13

Real-Time Workshop Pane	3-13
Comments Pane	3-17
Symbols Pane	3-20
Interface Pane	3-27
Templates Pane	3-35
Data Placement Pane	3-37
Data Type Replacement Pane	3-38
Optimization Pane	3-40
Tips for Optimizing the Generated Code	3-44
Use Auto-Optimized Targets	3-44
Use Configuration Wizard Blocks	3-44
Set Hardware Implementation Parameters Correctly	3-45
Remove Unnecessary Initialization Code	3-46
Generate Pure Integer Code If Possible	3-47
Disable MAT-File Logging	3-47
Use the Virtualized Output Ports Optimization	3-48
Use Stack Space Allocation Options	3-49
Using External Mode with the ERT Target	3-51
Generating a Code Generation Report	3-53
Automatic S-Function Wrapper Generation	3-56
Limitations	3-57
Generating an S-Function Wrapper	3-58

Custom Storage Classes

4

Introduction to Custom Storage Classes	4-3
Custom Storage Classes and Simulink Data Objects ..	4-5
Predefined CSCs	4-6
Setting the Custom Storage Class Properties	4-9
Generating Code with CSCs	4-10
Designing Custom Storage Classes	4-15
Custom Storage Class Designer Overview	4-15

Using the Custom Storage Class Designer	4-17
Creating Packages with CSC Definitions	4-30
Defining Advanced Custom Storage Class Types	4-34
Create Your Own Parameter and Signal Classes	4-34
Create a Custom Attributes Class for Your CSC (Optional)	4-34
Write TLC Code for Your CSC	4-35
Register Custom Storage Class Definitions	4-35
GetSet Custom Storage Class for Data Store Memory ..	4-38
Code Generation Example	4-39
Requirements and Restrictions for Use of CSCs	4-41
Setting Related Code Generation Options	4-41
Restrictions	4-41
Use of CSCs with Model Referencing	4-41
Older Custom Storage Classes (Prior to Release 14) ...	4-43
Simulink.CustomParameter Class	4-43
Simulink.CustomSignal Class	4-44
Instance Specific Attributes for Older Storage Classes ...	4-47
Assigning a Custom Storage Class to Data	4-49
Code Generation with Older Custom Storage Classes	4-49
Compatibility Issues for Older Custom Storage Classes ..	4-50

Advanced Code Generation Techniques

5

Introduction	5-3
Code Generation with User-Defined Data Types	5-5
Using User-Defined Data Types for Code Generation	5-7
Customizing the Target Build Process with the STF_make_rtw Hook File	5-8

File and Function Naming Conventions	5-8
STF_make_rtw_hook.m Function Prototype and Arguments	5-9
Applications for STF_make_rtw_hook.m	5-12
Using STF_make_rtw_hook.m for Your Build Procedure ..	5-13
Auto-Configuring Models for Code Generation	5-14
Utilities for Accessing Model Configuration Properties ...	5-14
Automatic Model Configuration Using ert_make_rtw_hook	5-15
Using the Auto-Configuration Utilities	5-17
Generating Efficient Code with Optimized ERT	
Targets	5-18
Default ERT Target	5-19
Optimized Fixed-Point ERT Target	5-19
Optimized Floating-Point ERT Target	5-21
Using the Optimized ERT Targets	5-23
Custom File Processing	5-26
Custom File Processing Components	5-26
Custom File Processing User Interface Options	5-27
Code Generation Template (CGT) Files	5-29
Using Custom File Processing (CFP) Templates	5-33
CFP Template Structure	5-34
Generating Source and Header Files with a CFP Template	5-35
Code Template API Summary	5-44
Generating Custom File Banners	5-47
Optimizing Your Model with Configuration Wizard	
Blocks and Scripts	5-53
Configuration Wizards vs. Auto-Configuring Targets	5-54
Adding a Configuration Wizard Block to Your Model	5-54
Using Configuration Wizard Blocks	5-57
Creating a Custom Configuration Wizard Block	5-57
Replacement of STF_rtw_info_hook Mechanism	5-65
Optimizing Task Scheduling for RTOS Targets	5-66
Using rtmStepTask	5-67

Suppressing the Redundant Scheduling Calls 5-68

Requirements, Restrictions, Target Files

6

Requirements and Restrictions 6-2

System Target File and Template Makefiles 6-4

Index

Getting Started

What Is Real-Time Workshop Embedded Coder? (p. 1-2)

Describes the features of Real-Time Workshop® Embedded Coder.

Real-Time Workshop Embedded Coder Feature Summary (p. 1-3)

Summary of the features of Real-Time Workshop Embedded Coder.

What You Need to Know to Use This Product (p. 1-5)

Prerequisite experience for use of Real-Time Workshop Embedded Coder; summary of related documentation.

Installing Real-Time Workshop Embedded Coder (p. 1-7)

Installation instructions.

Real-Time Workshop Embedded Coder Demos (p. 1-8)

Information on interactive demos and example code provided to help you learn about Real-Time Workshop Embedded Coder.

What Is Real-Time Workshop Embedded Coder?

Real-Time Workshop Embedded Coder is a separate, add-on product for use with Real-Time Workshop. It is intended for use in embedded systems development. Real-Time Workshop Embedded Coder generates code that is easy to read, trace, and customize for your production environment.

Real-Time Workshop Embedded Coder provides a framework for the development of production code that is optimized for speed, memory usage, and simplicity. Real-Time Workshop Embedded Coder generates optimized ANSI-C or ISO-C code for fixed-point and floating-point microprocessors. It extends the capabilities provided by Real-Time Workshop to support specification, integration, deployment, and testing of production applications on embedded targets. Real-Time Workshop Embedded Coder addresses targeting considerations such as RAM, ROM, and CPU constraints, code configuration, and code verification.

The Embedded Real-Time (ERT) target provided by Real-Time Workshop Embedded Coder is designed for customization. Most users want to generate code for a particular microprocessor or development board, and to deploy the code on target hardware with a cross-development system. To do this, some modifications to the ERT target files are required. This document and its companion, the Developing Embedded Targets document, describe how to customize the ERT target for your production requirements.

For large-scale, multi-model projects involving teams of engineers, Real-Time Workshop Embedded Coder offers Module Packaging Features (MPF) you can use to control the number and organization of files generated, the location of global identifiers, registration of user-defined data types, customized comments, and the location of target variables.

Real-Time Workshop Embedded Coder Feature Summary

In addition to supporting the features of Real-Time Workshop, Real-Time Workshop Embedded Coder:

- Generates ANSI/ISO C or C++ code and executables from Simulink® and Stateflow® models with memory usage, execution speed, and readability comparable to handwritten code
- Extends Real-Time Workshop and Stateflow Coder with the optimizations and code configuration features essential for production deployment
- Supports all Simulink data objects and data dictionary capabilities, including user-defined storage classes, types, and aliases
- Provides an intuitive graphical user interface for creating custom data
- Concisely partitions multirate code for efficient scheduling with or without a real-time operating system (RTOS)
- Provides a rich set of commenting capabilities to trace code to models and requirements
- Verifies code by automatically importing it into Simulink for software-in-the-loop testing
- Generates code documentation that is integrated with the Simulink Model Explorer and hyperlinked to the model
- Provides a Model Advisor that checks your model configuration and offers advice on how to optimize or tune a configuration set based on your stated goals or style.
- Generates an extensible main program based on information you provide for deploying the code in your real-time environment
- Generates single-rate or multirate code using periodic sample times specified in a model
- Applies a strategy called rate grouping for multi-rate, multitasking models, which generates separate functions for the base rate task and for each sub-rate task in the model
- Provides an option to easily transition between the Real-Time Workshop generic real-time (GRT) target and the Real-Time Workshop Embedded Coder embedded real-time (ERT) target

- Provides extensible module packaging features that let you package generated code to comply with specific software styles and standards
- Provides capabilities for verifying generated code, including the ability to import generated code back into Simulink as an S-function for software-in-the-loop testing with a plant model, generation of user-controlled comments and descriptions to improve code readability and traceability, inclusion of requirements in generated code, and persistent identifier names for minimizing code differences between model revisions
- Documents generated code in an HTML report that comprehensively describes code modules and model configuration settings applied during code generation
- Supports international (non-US-ASCII) characters encountered during code generation when found in Simulink block names and block descriptions, user comments on Stateflow diagrams, Stateflow object descriptions, custom TLC files, and code generation template files. For details about international character support, see [Support for International \(Non-US-ASCII\) Characters in the Real-Time Workshop documentation](#).

What You Need to Know to Use This Product

Prerequisites

To use Real-Time Workshop Embedded Coder, you should have basic familiarity with MATLAB®, Simulink, and Real-Time Workshop. If you have not done so, you should read:

- The tutorials in the document Getting Started with Real-Time Workshop. The tutorials provide hands-on experience in configuring models for code generation and generating code.
- The “Program Architecture” and “Models with Multiple Sample Rates” chapters of the Real-Time Workshop documentation. These sections give a general overview of the architecture and execution of programs generated by Real-Time Workshop.

Real-Time Workshop Embedded Coder Documentation Collection

The Real-Time Workshop Embedded Coder documentation collection consists of the following:

Document	Description
User’s Guide	Describes ERT model execution, timing, and task management; explains how to interface to and call model code; describes default ERT code generation options; and discusses advanced configuration options.
Module Packaging Features	Explains how to use the Module Packaging Features.
Developing Embedded Targets	Describes requirements and implementation details for creating custom embedded targets based on the supplied ERT.

Related Documentation

You may be interested in the following documentation, especially if you are planning to implement custom embedded targets:

Document	Description
Real-Time Workshop User's Guide: Writing S-Functions for Real-Time Workshop	Discusses inlining and code generation issues relevant to device drivers and other S-functions
Real-Time Workshop User's Guide: Data Exchange APIs	Explains how to interface signals and parameters within generated code to your own code; combine code generated from multiple models into a single system; and implement external mode communication with your own low-level protocol layer.
Target Language Compiler	Provides details about the Target Language Compiler (TLC) needed to make non-trivial modifications to the ERT system target file, use TLC hooks into the build process, utilize information from the <i>model.rtw</i> file, implement inlined device drivers, or pass information into or out of the TLC phase of the build process.
Writing S-Functions	Explains how to write fully inlined S-functions. This information is necessary for developing device driver blocks for a target.

Installing Real-Time Workshop Embedded Coder

Your platform-specific MATLAB installation documentation provides all of the information you need to install Real-Time Workshop Embedded Coder.

Prior to installing Real-Time Workshop Embedded Coder, you must obtain a License File or Personal License Password (PLP) from The MathWorks. The License File or PLP identifies the products you are permitted to install and use.

If you customize your installation, the installer displays a dialog box that lets you select the MATLAB products to install. You can select and install only products for which you are licensed.

Real-Time Workshop Embedded Coder has product prerequisites, described in the following table, that must be met for proper installation and execution.

Licensed Product	Prerequisite Products	Additional Information
Simulink	MATLAB 7 (Release 14)	—
Real-Time Workshop	Simulink 6 (Release 14)	Requires Borland C, LCC, Visual C/C++, or Watcom C compiler to create MATLAB MEX-files on your platform.
Real-Time Workshop Embedded Coder	Real-Time Workshop 6 (Release 14)	—

If you experience installation difficulties and have Web access, use the resources available on the Mathworks Web site Installation and Licensing page at <http://www.mathworks.com/support/install.html>.

Real-Time Workshop Embedded Coder Demos

The Real-Time Workshop demo suite contains many demos that can help you become familiar with features of Real-Time Workshop Embedded Coder and to inspect generated code. These demos illustrate features specific to Real-Time Workshop Embedded Coder and also general Real-Time Workshop features as used with Embedded Coder.

If you are reading this document online in the MATLAB Help browser, you can open the demo suite by clicking on this link: [rtwdemos](#)

Alternatively, you can access the demo suite by typing the name of the demo library at the MATLAB command prompt:

```
rtwdemos
```

Most of the demos provide a button titled **Generate Code Using Real-Time Workshop Embedded Coder**. When you click this button, the demo auto-configures itself for code generation using the ERT target, and then initiates the code generation process. If your installation is licensed for Real-Time Workshop Embedded Coder, use this button.

If your installation is not licensed for Real-Time Workshop Embedded Coder, you can run most of the demos by clicking on the **Generate Code Using Real-Time Workshop** button. When you click this button, the demo auto-configures itself for code generation using the GRT target, and then initiates the code generation process. Note that the GRT target provides a subset of the capabilities of the ERT target.

Data Structures and Program Execution

Data Structures and Code Modules (p. 2-2)	Main data structures, code modules, and header files of Real-Time Workshop Embedded Coder.
Program Execution (p. 2-10)	Overview of Real-Time Workshop Embedded Coder generated programs.
Stand-Alone Program Execution (p. 2-11)	Execution and task management in stand-alone (bare board) generated programs.
VxWorks Example Main Program Execution (p. 2-20)	Data Exchange Execution and task management of example programs deployed under VxWorks real-time operating system.
Model Entry Points (p. 2-22)	Description of model entry-point functions and how to call them.
The Static Main Program Module (p. 2-26)	Description of the alternative static (non-generated) main program module.
Rate Grouping Compliance and Compatibility Issues (p. 2-31)	How to take advantage of the efficiency of rate grouping by updating your multi-rate inlined S-functions and main program module for compatibility.

Data Structures and Code Modules

Real-Time Model Data Structure

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

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_Tag {  
    ...  
};
```

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

```
typedef struct _RT_MODEL_model_Tag 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`, Real-Time Workshop Embedded Coder 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 “Generating a Code Generation Report” on page 3-53). Then, view `model.h` by clicking the hyperlink in the report.

Code Modules

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 “Generating a Code Generation Report” on page 3-53 for further information.

Generated Code Modules

Real-Time Workshop Embedded Coder 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 Real-Time Workshop Embedded Coder.

Note The file packaging of Real-Time Workshop Embedded Coder 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 all code implementing the model algorithm (for example, <i>model_step</i> , <i>model_initialize</i> , <i>model_terminate</i> , <i>model_SetEventsForThisBaseStep</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 extern in <i>model.h</i> .
<i>model_types.h</i>	Provides forward declarations for the real-time model data structure and the parameters data structure. 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" 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 code only and Generate an example main program options are off. (See “Generating the Main Program” on page 2-8.)</p> <p>autobuild.h contains #include directives required by the static version of the ert_main.c main program module. Since the static ert_main.c is not created at code generation time, it includes autobuild.h to access model-specific data structures and entry points.</p> <p>See “The Static Main Program Module” on page 2-26 for further information.</p>
<p>model_capi.c or .cpp model_capi.h (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 model_capi.c or .cpp and .h 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 Real-Time Workshop 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 4, “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

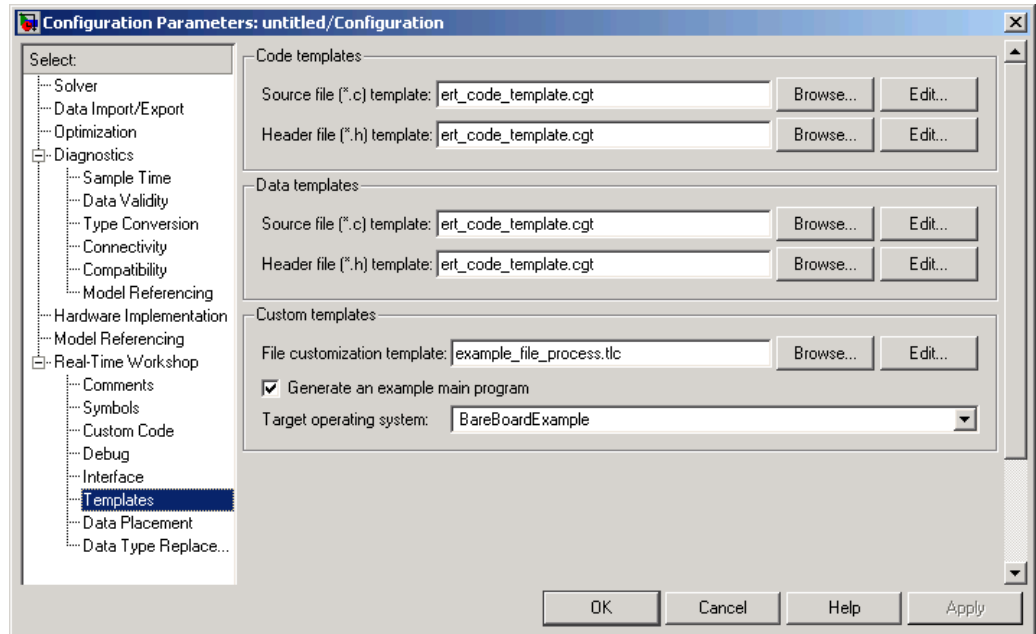
Code that you write to interface with generated model code usually includes a customized main module (based on a main program provided by Real-Time Workshop Embedded Coder), and may also include interrupt handlers, device driver blocks and other S-functions, and other supervisory or supporting code.

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

The **Generate an example main program** option controls whether or not `ert_main.c` or `ert_main.cpp` is generated. This option is located in the **Templates** pane of the Configuration Parameters dialog, 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 bare-board 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 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 multi-rate, and also upon your model's solver mode (`SingleTasking` vs. `MultiTasking`). These are described in detail in “Program Execution” on page 2-10.

If you turn the **Generate an example main program** option off, Real-Time Workshop Embedded Coder provides a static version of the file `ert_main.c` as a basis for your custom modifications (see “The 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 5-40.

Program Execution

The following sections describe how programs generated by Real-Time Workshop Embedded Coder 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.
- “VxWorks Example Main Program Execution” on page 2-20 describes the operation of example programs designed for deployment under the VxWorks real-time operating system.
- “Model Entry Points” on page 2-22 describes the model functions that are generated for both stand-alone and VxWorks example programs.

Stand-Alone Program Execution

By default, Real-Time Workshop Embedded Coder 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 ERT code generation options (3) category of the **Real-Time Workshop** tab of the Simulation Parameters dialog, 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 multi-rate. In a single-rate model, `rt_OneStep` simply calls the `model_step` function. In a multi-rate model, `rt_OneStep` prioritizes and schedules execution of blocks according to the rates at which they run.

Real-Time Workshop Embedded Coder generates significantly different code for multi-rate 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

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

The operation of `rt_OneStep` depends upon

- Whether your model is single-rate or multi-rate. 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 multi-rate.
- 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 multi-rate 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	Multi-Rate
SingleTasking	Allowed	Allowed
MultiTasking	Disallowed	Allowed
Auto	Allowed (defaults to SingleTasking)	Allowed (defaults to MultiTasking)

The generated code for `rt_OneStep` (and associated timing data structures and support functions) is tailored to the number of rates in the model and to the solver mode. The following sections discuss each possible case.

Single-Rate 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.

Multi-Rate Multitasking Operation

In a multi-rate multitasking system, Real-Time Workshop Embedded Coder 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 multi-rate 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 multi-rate, multitasking models, Real-Time Workshop Embedded Coder 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 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 with the `model_SetEventsForThisBaseStep` function. When a counter indicates that a task's sample period has elapsed, `model_SetEventsForThisBaseStep` 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. Multi-rate `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 multi-rate 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 multi-rate, multitasking models generate more efficient code.

Multi-Rate Singletasking Operation

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

In a multi-rate 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 multi-rate 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 multi-rate 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`. `model_step` 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`.
- In a multi-rate, multitasking model, set model inputs associated with sub-rates before calling `model_stepN` in the sub-rate loop.
- In a multi-rate, 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 multi-rate `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.

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.

VxWorks Example Main Program Execution

Overview

The Real-Time Workshop Embedded Coder VxWorks example main program is provided 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 VxWorks is required. See your VxWorks documentation.

To generate a VxWorks example program:

- 1** In the ERT code generation options (3) category of the **Real-Time Workshop** tab of the Simulation Parameters dialog, 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

In a VxWorks example program, the main program and the base rate and sub-rate tasks (if any) run as prioritized tasks under VxWorks. 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 VxWorks, rather than by a timer interrupt. On each activation, *tBaseRate* calls *model_step*.

Note that the clock rate granted by VxWorks may not be the same as the rate requested by *model_main*.

Multi-Rate Multitasking Operation

In a multi-rate, 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 multi-rate, multitasking models. The base rate task calls *model_step0*, while the sub-rate tasks call *model_stepN*. The base rate task is responsible for maintaining event flags and scheduling counters, using the same rate monotonic scheduler algorithm as a stand-alone program.

Multi-Rate Singletasking Operation

In a multi-rate, 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 VxWorks, 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

This section discusses the entry points to the generated code.

Note that the calling interface generated for each of these functions differs significantly depending on how you set the **Generate reusable code** option (See “Interface Pane” on page 3-27).

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 descriptions below document the default (**Generate reusable code** off) calling interface generated for these functions.

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.

model_step

Default Calling Interface

The *model_step* function prototype is different depending upon the number of rates in the model and the solver mode. Function Prototypes for *model_step* on page 2-23 shows the *model_step* function prototype for each case.

Function Prototypes for `model_step`

Rates/Solver Mode	Function Prototype	Arguments
Single-rate/SingleTasking	<code>void model_step(void);</code>	N/A
Multi-rate/SingleTasking	<code>void model_step(int_T tid);</code>	<code>tid</code> is a task identifier.
Multi-rate/MultiTasking (rate grouping)	<code>void model_stepN (void);</code> (<i>N</i> is a task identifier)	N/A

Operation

`model_step` combines the model output and update functions into a single routine. `model_step` is designed to be called at interrupt level from `rt_OneStep`, which is assumed to be invoked as a timer ISR.

See “`rt_OneStep`” on page 2-13 for a description of how calls to `model_step` are generated and scheduled for the above cases.

`model_step` computes the current value of all blocks. If logging is enabled, `model_step` updates logging variables. If the model’s stop time is finite, `model_step` signals the end of execution when the current time equals the stop time.

In cases where a `tid` is passed in, the caller (`rt_OneStep`) assigns each task a `tid`, and `model_step` uses the `tid` argument to determine which blocks have a sample hit (and therefore should execute).

Under any of the following conditions, `model_step` does not check the current time against the stop time:

- The model’s stop time is set to `inf`.
- Logging is disabled.
- The **Terminate function required** option is not selected.

Therefore, if any of these conditions are true, the program runs indefinitely.

model_initialize

Default Calling Interface

The *model_initialize* function prototype is

```
void model_initialize(boolean_T firstTime);
```

Operation

If *firstTime* equals 1 (TRUE), *model_initialize* initializes *rtModel* and other data structures private to the model. If *firstTime* equals 0 (FALSE), *model_initialize* resets the model's states, but does not initialize other data structures.

The generated code calls *model_initialize* once, passing in *firstTime* as 1(TRUE).

model_terminate

Default Calling Interface

The *model_terminate* function prototype is

```
void model_terminate(void);
```

Operation

When *model_terminate* is called, blocks that have a terminate function execute their terminate code. If logging is enabled, *model_terminate* ends data logging. *model_terminate* should only be called once. If your application runs indefinitely, you do not need the *model_terminate* function.

If you do not require a terminate function, see “Interface Pane” on page 3-27 for information on using the **Terminate function required** option. Note that if **Terminate function required** is off, the program runs indefinitely.

model_SetEventsForThisBaseStep

Calling Interface

By default, the *model_SetEventsForThisBaseStep* function prototype is

```
void model_SetEventsForThisBaseStep(boolean_T *eventFlags)
```

where *eventFlags* is a pointer to the model's event flags array.

If **Generate reusable code** is on, an additional argument is included:

```
void model_SetEventsForThisBaseStep(boolean_T *eventFlags,  
                                     RT_MODEL_model *model_M);
```

where *model_M* is a pointer to the real-time model object.

Operation

The *model_SetEventsForThisBaseStep* function is a utility function that is generated and called only for multi-rate, multitasking programs.

model_SetEventsForThisBaseStep maintains the event flags, which determine which sub-rate tasks need to run on a given base rate time step. *model_SetEventsForThisBaseStep* must be called prior to calling the *model_step* function. See “Multi-Rate Multitasking Operation” on page 2-21 for further information.

Note The macro `MODEL_SETEVENTS`, defined in the static `ert_main.c` module, provides a way to call *model_SetEventsForThisBaseStep* from a static main program.

The Static Main Program Module

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” 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 multi-rate, 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 multi-rate, 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 multi-rate, 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 2-12 and “Guidelines for Modifying `rt_OneStep`” on page 2-18.

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 multi-rate 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.
- When the **Generate code only** and **Generate an example main program** options are off, Real-Time Workshop Embedded Coder 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

Main Program Compatibility

When the **Generate an example main program** option is off, Real-Time Workshop Embedded Coder 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 blocks in the Signal Processing Blockset, are compliant with the requirements for generating rate grouping code. However, user-written multi-rate 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 multi-rate 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..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
(p. 3-3)

GUIs for viewing and configuring ERT target options.

Support for Continuous Time Blocks and Solvers (p. 3-5)

Summarizes ERT options in relation to trade-offs for code efficiency, traceability, and safety.

Mapping Application Requirements to Configuration Options (p. 3-6)

Discusses the process of mapping application requirements, particularly with respect to traceability, efficiency, and safety, map to code generation options in a model configuration set.

A Guide to the ERT Target Options
(p. 3-13)

Describes code generation options that are specific to the ERT target.

Tips for Optimizing the Generated Code (p. 3-44)

Utilities and code generation options you can use to automatically configure models, improve performance and reduce code size.

Generating a Code Generation Report (p. 3-53)

Describes how to generate a report including information on the generated code and suggestions for optimization. You can view the report in any HTML browser. The report includes hyperlinks from the generated code to the source blocks in your model.

Automatic S-Function Wrapper Generation (p. 3-56)

How to integrate your Real-Time Workshop Embedded Coder code into a model by generating S-function wrappers.

Accessing the ERT Target Options

This chapter describes the Embedded Real-Time (ERT) target code generation options, and how to view and configure them. 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 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 Embedded Target for Motorola MPC555 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 Model Explorer and the Configuration Parameters dialog. The *Using Simulink* document contains detailed information on these topics.
- The general 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.

Viewing ERT Target Options in the Configuration Parameters Dialog

The Configuration Parameters dialog provides the quickest route to a model's active configuration set. Illustrations throughout this chapter show the Configuration Parameters view of model parameters (except as otherwise noted).

“A Guide to the ERT Target Options” on page 3-13 discusses each category of ERT target options displayed in the panes of the Configuration Parameters dialog. See the following sections for information on how to use the various categories of options.

- “Real-Time Workshop Pane” on page 3-13
- “Comments Pane” on page 3-17
- “Symbols Pane” on page 3-20
- “Interface Pane” on page 3-27
- “Templates Pane” on page 3-35
- “Data Placement Pane” on page 3-37
- “Data Type Replacement Pane” on page 3-38
- “Optimization Pane” on page 3-40

Support for Continuous Time Blocks and Solvers

Continuous Block Support

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 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 “Recommended for Production Code?” column of the table.

Continuous Solver Support

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.

Mapping Application Requirements to Configuration Options

The first step to applying Real-Time Workshop Embedded Coder 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 Simulink Configuration Parameters dialog 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 trade-offs can be made?

Once you have answered these questions, review Mapping of Application Requirements to Configuration Parameters on page 3-7. This table maps requirements of traceability, efficiency, and safety to configuration options that are available for the Embedded Real-Time (ERT) target.

The default settings that appear in the table are default factory settings.

Mapping of Application Requirements to Configuration Parameters

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Optimization					
Block reduction optimization	No impact	Clear	Set	No impact	Set
Implement logic signals as boolean data (vs. double)	No impact	No impact	Set	No impact	Set
Inline parameters	Set	Set	Set	No impact	Clear
Conditional input branch execution	No impact	Set	Set	No impact	Set
Signal storage reuse	Clear	Clear	Set	No impact	Set
Application lifespan (days)	No impact	No impact	Set to correct value	No impact	1
Enable local block outputs	Clear	No impact	Set	No impact	Set
Ignore integer downcasts in fold expressions	Clear	No impact	Set	Clear	Clear
Eliminate superfluous temporary variables (Expression folding)	Clear	Clear	Set	No impact	Set
Loop unrolling threshold	No impact	No impact	>0	No impact	5
Reuse block outputs	Clear	Clear	Set	No impact	Set

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Inline invariant signals	Clear	Clear	Set	No impact	Set
Remove root level I/O zero initialization	No impact	No impact	Set	Clear	Clear
Remove internal state zero initialization	No impact	No impact	Set	Clear	Clear
Use memset to initialize floats and doubles to 0.0	No impact	No impact	Set	Clear	Clear
Optimize initialization code for model reference	No impact	No impact	Set	Clear	Set
Remove code that protects against division arithmetic exceptions (fixed-point)	No impact	No impact	Set	Clear	Clear
Hardware Implementation					
Number of bits	No impact	No impact	Set	No impact	8, 16, 32, 32
Signed integer division rounds to	Undefined	Zero or Floor	Zero	Floor	Undefined
Real-Time Workshop					
Generate HTML report	Set	Set	No impact	No impact	Clear
Include hyperlinks to model	Set	Set	No impact	No impact	Clear

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Launch report after code generation completes	Set	Set	No impact	No impact	Clear
Ignore custom storage classes	No impact	No impact	No impact	No impact	Clear
Real-Time Workshop: Comments					
Include comments	Set	Set	No impact	No impact	Set
Simulink block comments	Set	Set	No impact	No impact	Set
Show eliminated statements	No impact	Set	No impact	No impact	Clear
Verbose comments for Simulink Global storage class	Set	Set	No impact	No impact	Clear
Simulink block descriptions	Set	Set	No impact	No impact	Clear
Simulink data object descriptions	Set	Set	No impact	No impact	Clear
Custom comments (MPT objects only)	Set	Set	No impact	No impact	Clear
Stateflow object descriptions	Set	Set	No impact	No impact	Clear
Requirements in block comments	Set	Set	No impact	No impact	Clear
Real-Time Workshop: Symbols					
Symbol format	No impact	Set	No impact	No impact	\$R\$N\$M

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Minimum mangle length	No impact	1	No impact	No impact	1
Maximum identifier length	Set	>30	No impact	No impact	31
Generate scalar inlined parameters as	No impact	Macros	Literals	No impact	Literals
#define naming	No impact	Force uppercase	No impact	No impact	None
Parameter naming	No impact	Force uppercase	No impact	No impact	None
Signal naming	No impact	Force uppercase	No impact	No impact	None
Real-Time Workshop: Debug					
Verbose builds	Set	No impact	No impact	Set	Set
Retain .rtw file	Set	Set	No impact	No impact	Clear
Real-Time Workshop: Interface					
Target floating-point math environment	No impact	No impact	Set	No impact	ANSI-C
Utility function generation	No impact	No impact	Shared	No impact	Auto
Support floating-point numbers	No impact	No impact	Clear for integer only	No impact	Set
Support complex numbers	No impact	No impact	Clear for real only	No impact	Set

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Support non-finite numbers	No impact	No impact	Clear	No impact	Set
Support absolute time	No impact	No impact	Clear	No impact	Set
Support continuous time	No impact	No impact	Clear	No impact	Clear
Support non-inlined S-functions	No impact	No impact	Clear	No impact	Clear
Terminate function required	No impact	No impact	Clear	Set	Set
Generate reusable code	No impact	No impact	Set for single instance	No impact	Clear
Suppress error status in real-time model data structure	Clear	No impact	Set	Clear	Clear
Single update/output function	Set	Set	Set	No impact	Set
GRT compatible call interface	No impact	Clear	Clear	No impact	Clear
Create Simulink (S-Function) block	Set	No impact	No impact	No impact	Clear
MAT-file logging	Set	No impact	Clear	No impact	Clear
Real-Time Workshop: Data Placement					
Data definition	No impact	Set	No impact	No impact	Auto

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Data declaration	No impact	Set	No impact	No impact	Auto
Module naming	No impact	Set	No impact	No impact	Not specified
Signal display level	No impact	Set	No impact	No impact	10
#include file delimiter	No impact	Set	No impact	No impact	Auto
Parameter tune level	No impact	Set	No impact	No impact	10
Source of initial values	No impact	Set	No impact	No impact	Model

A Guide to the ERT Target Options

This section describes options that are specific to the ERT target, as they appear in the Configuration Parameters dialog.

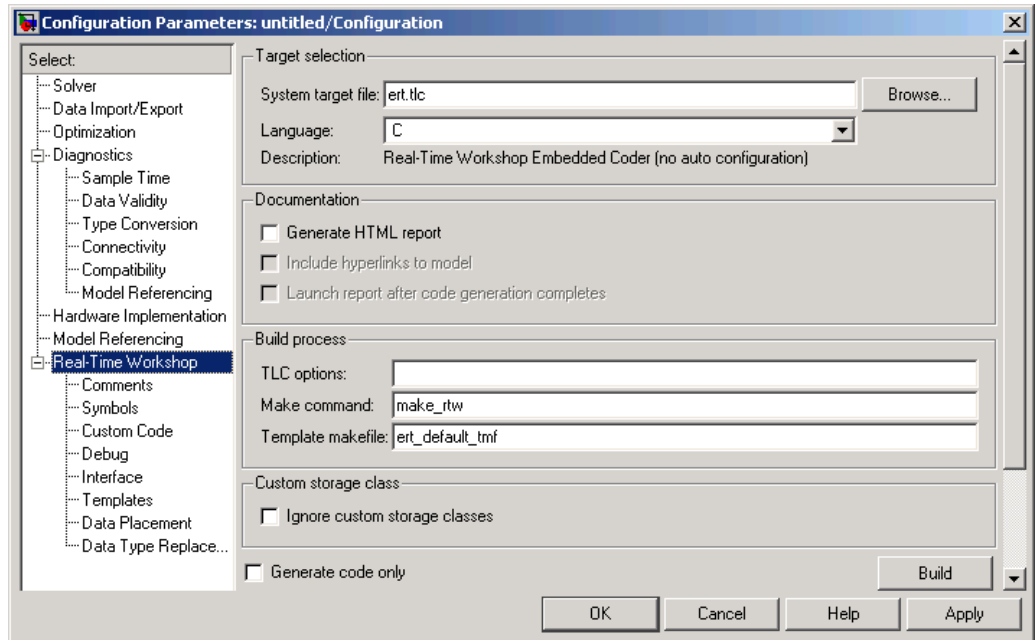
Some panes of the Configuration Parameters dialog (for example, the **Templates** and **Interface** panes) contain only ERT-specific options. Others (for example, the **Real-Time Workshop** pane) display a combination of general Real-Time Workshop options and ERT target options. The discussion below focuses on the ERT-specific options, with references to related options and documentation given as necessary.

In the illustrations below, options are shown set to their default values (unless noted otherwise).

Real-Time Workshop Pane

The **Real-Time Workshop** pane contains general code generation options that pertain to

- System target file selection
- Target language selection
- Report generation
- Options and files to be used in the build process
- The use of custom storage classes
- Whether to generate code only or complete the entire build process



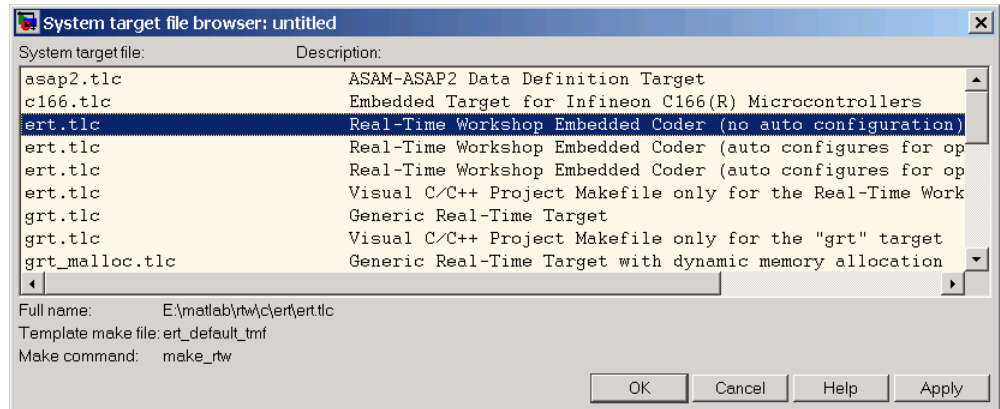
Target Selection Subpane

The **Browse** button in the **Target Selection** subpane lets you select a target with the System Target File Browser. See 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, Real-Time Workshop Embedded Coder provides three variants of the ERT target. These are

- **Optimized fixed-point ERT target:** Select this target to generate code with automatic configuration of options that are optimized for fixed-point code generation.
- **Optimized floating-point ERT target:** Select this target to generate code with automatic configuration of options that are optimized for floating-point code generation.
- **Default ERT target:** Does not automatically configure any options. 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 5-18.

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 5-14 discusses the auto-configuration mechanism and utilities used by the optimized ERT target variants.

Use the **Language** menu in the **Target selection** subpane to select the target language for the code Real-Time Workshop generates. You can select C or C++. Real-Time Workshop generates `.c` or `.cpp` files, depending on your selection, and places the files in your build directory.

Note If you select C++, you might need to configure Real-Time Workshop to use the appropriate compiler before you build a system. For details, see “Choosing and Configuring a Compiler” in the Real-Time Workshop documentation.

Documentation Subpane

Options in the **Documentation** subpane control generation of the extended Real-Time Workshop Embedded Coder HTML code generation report. Options are

- **Generate HTML Report:** When this option is selected, the code generation process generates an HTML code generation report, as described in “Generating a Code Generation Report” on page 3-53. Selecting this option enables the two related options immediately below it.

By default, **Generate HTML Report** is deselected.

- **Include hyperlinks to model:** When you select this option, the HTML report includes hyperlinks from the code to the generating blocks in the model. By deselecting this option, you can speed up code generation. For very large models (containing over 1000 blocks) generation of hyperlinks can be time consuming.

This option is enabled and selected when **Generate HTML Report** is selected.

- **Launch report after code generation completes:** When you select this option, the HTML report is automatically displayed in a MATLAB Web browser window after code generation. If you prefer not to have the browser come to the front after code generation, deselect this option.

This option is enabled and selected when **Generate HTML Report** is selected.

Build Process Subpane

The options in the **Build process** subpane are described in the Real-Time Workshop documentation.

For examples of how arguments in the **Make command** and **TLC options** fields are passed to the build process, see:

- “Customizing the Target Build Process with the STF_make_rtw Hook File” on page 5-8
- The “Understanding and Using the Build Process” section of the Developing Embedded Targets document

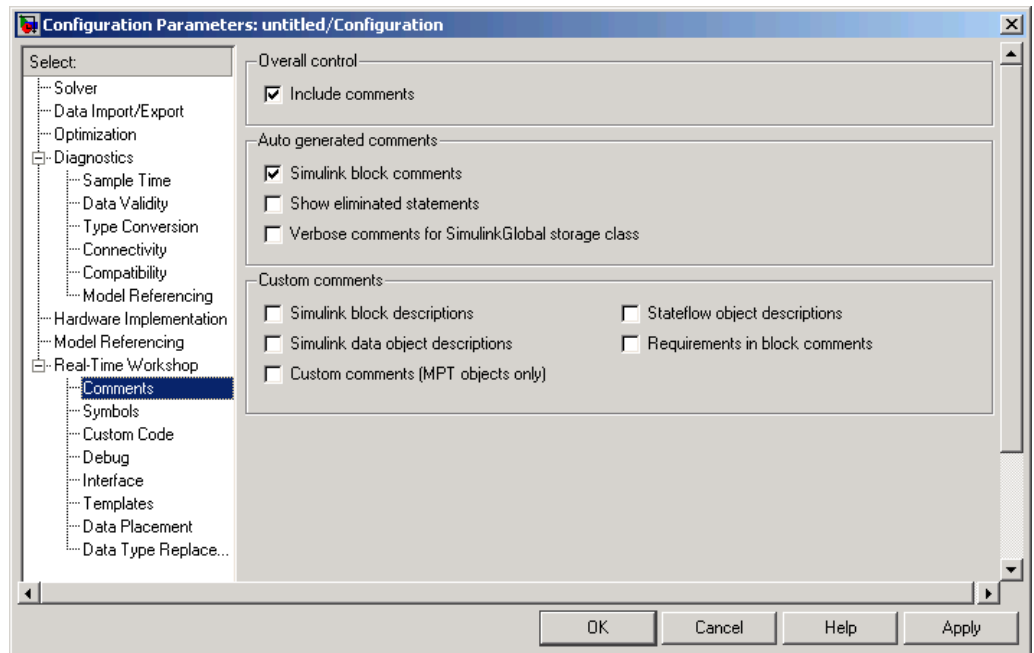
Custom Storage Class Subpane

If you have defined data objects with custom storage classes in your model for use with Real-Time Workshop Embedded Coder, you should make sure that the **Ignore custom storage classes** option is deselected.

Chapter 4, “Custom Storage Classes” contains a detailed description of the use of custom storage classes in code generation.

Comments Pane

The **Comments** pane contains options related to generation of comments in generated code.



Overall Control Subpane

The **Include comments** option in the **Overall control** subpane enables or disables all other options on the **Comments** pane.

Auto Generated Comments Subpane

The **Auto generated comments** subpane contains options that are common to all targets. See the Real-Time Workshop documentation for information on the other options in the **Auto generated comments** subpane.

Custom Comments Subpane

The **Custom comments** subpane supports options that are specific to the ERT target. These options let you enable or suppress generation of descriptive information in comments for blocks and other objects in the model. These options are

- **Simulink block descriptions:** You can enter descriptive information for any block in the **Description** field of the Block Properties dialog. When you select **Simulink block descriptions**, the code generator:
 - Includes the text specified in the **Description** field in the comments for the code generated for each block
 - Adds a comment that includes the block name at the start of the code for each block, regardless of whether you select **Simulink block comments**

The block description text 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.)

Note For virtual blocks or blocks that have been removed due to block reduction optimizations, no comments are generated.

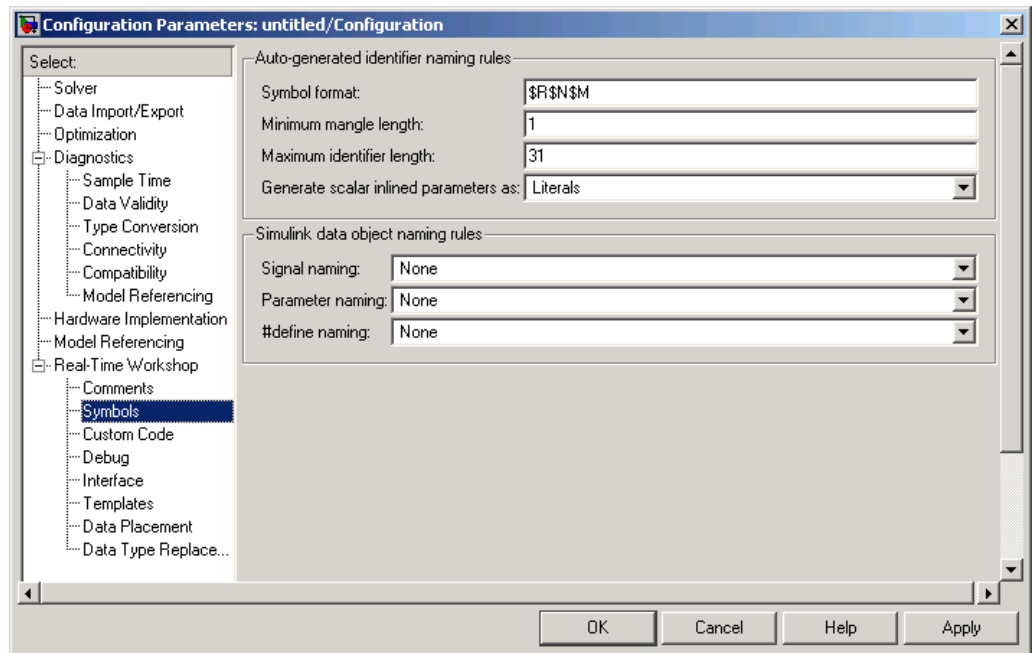
- **Simulink data object descriptions:** You can enter descriptive information for Simulink data objects (such as signal, parameter, data type, and bus objects) with the **Description** field of the object properties in the Simulink Model Explorer. When the **Simulink data object descriptions** option is selected, the **Description** text is included in comments in code generated for each object.
- **Custom comments (MPT objects only):** You can include comments just above signals and parameter identifiers in the generated code as specified

in an M-code or TLC function. See the Module Packaging Features document for more information.

- **Stateflow object descriptions:** You can enter descriptive information for any Stateflow state, chart, transition, or graphical function in the **Description** field of the Properties dialog of the Stateflow object. When you select **Stateflow object descriptions**, the code generator includes the **Description** text in comments that appear just above the code generated for each object, including any 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.)
- **Requirements in block comments:** When you select **Requirements in block comments**, the code generator includes the requirements that you 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.

Symbols Pane

The **Symbols** pane contains options that control the generation of symbols (such as variable names) in generated code. Most of these options are specific to the ERT target. Some **Symbols** pane options are common to all targets; these are described in the Real-Time Workshop documentation.



Auto-Generated Identifier Naming Rules Subpane

- **Symbol format:** Lets you customize generated symbols. You can enter a macro string that specifies whether, and in what order, certain substrings are included within generated symbols. This option affects the generation of symbols for
 - Signals and parameters that have Auto storage class.
 - Subsystem function names that are not user defined.
 - All Stateflow names.

For details on how to specify formats, see “Specifying Symbol Formats” on page 3-22

- **Minimum mangle length:** See “Name Mangling” on page 3-23.
- **Maximum identifier length:** Specifies the maximum number of characters (default 31) in generated function, typedef, and variable names. If you expect your model to generate lengthy symbols (due to use of long signal or parameter names, for example), or you find that symbols are being mangled more than expected, you should increase the **Maximum identifier length**.

Note that the **Maximum identifier length** interacts with the **Symbol format** specification, as described below.

- **Generate scalar inlined parameters as:** This option takes effect when the **Inline parameters** option is selected. For scalar inlined parameters, this menu lets you control how parameter values are expressed in the generated code. You can specify one of the following:
 - **Literals:** Parameters are expressed as numeric constants. This is the default, and is backward compatible with prior versions of Real-Time Workshop that did not support this option. Use of **Literals** can help in debugging TLC code, as it makes the values of parameters easy to search for.
 - **Macros:** Parameters are expressed as variables (with **#define** macros). The **Macros** option can make code more readable.

Simulink Data Object Naming Rules Subpane

- **Signal naming:** Use this option to define rules that change the names of a model’s signals.
- **Parameter naming:** Use this option to define rules that change the names of all of a model’s parameters.
- **#define naming:** Use this option to define rules that change the names of a model’s parameters that have a storage class of **Define**.

For more information on these options, see the Module Packaging Features document.

Specifying Symbol Formats

The **Symbol format** option lets you customize generated symbols by entering a macro string that specifies whether, and in what order, certain substrings are included within generated symbols. For example, you can specify that the model name be inserted into each symbol.

The macro string can include

- Tokens of the form \$X, where X is a single character. Valid tokens are listed in Symbol Format Tokens on page 3-22. You can use or omit tokens as you want, with the exception of the \$M token, which is required (see “Name Mangling” on page 3-23 below). You can place tokens in any order.
- Any valid C or C++ language identifier characters (a-z, A-Z, _ , 0-9).

The build process generates each symbol by expanding tokens (in the order listed in Symbol Format Tokens on page 3-22) and inserting the resultant strings into the symbol. Character strings between tokens are simply inserted directly into the symbol. Contiguous token expansions are separated by the underscore (_) character.

Symbol Format Tokens

Token	Description
\$M	Insert name mangling string, if required to avoid naming collisions (see “Name Mangling” on page 3-23). Note: This token is required.
\$R	Insert root model name into identifier. Note that when using model referencing, this token is required in addition to \$M (see “Model Referencing Considerations” on page 3-26). Note: This token replaces the Prefix model name to global identifiers option used in previous releases.
\$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.

Symbol 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 Simulink.</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.</p> <p>Note: This token replaces the Include data type acronym in identifier option used in previous releases.</p>

The default **Symbol format** specification is `RN$M`. 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 that is generated (if required) to resolve potential conflicts with other generated symbols (see “Name Mangling” on page 3-23 below).

Non-ERT based targets (such as the GRT target) use the default (`RN$M`) specification implicitly.

Name Mangling

In symbol generation, a circumstance that would cause generation of two or more identical symbols 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 symbols. Each name mangling string is guaranteed to be unique for each generated identifier.

The position of the `$M` token in the **Symbol format** specification determines the position of the name mangling string in the generated symbols. For example, if the default specification (`RN$M`) is used, the name mangling string is appended (if required) to the end of the symbol.

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 symbols 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 symbol names that appear in generated code. There are two ways of achieving this in Real-Time Workshop embedded Coder:

- 1 Choose unique names for objects in Simulink (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 **Symbol format** 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 3-24 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

How Name Mangling Strings Are Computed (Continued)

Object Type	Source of Mangling String
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 symbol before the mangle length is increased.

Minimizing Name Mangling

Note that the length of generated symbols 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 Symbol Format Tokens on page 3-22. 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 symbols 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 **Symbol format** specification (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 a symbol within the scope of a higher-level model and a symbol within the scope of a referenced model, the symbol from the referenced model is preserved. Name mangling is performed on the symbol from the higher-level model.

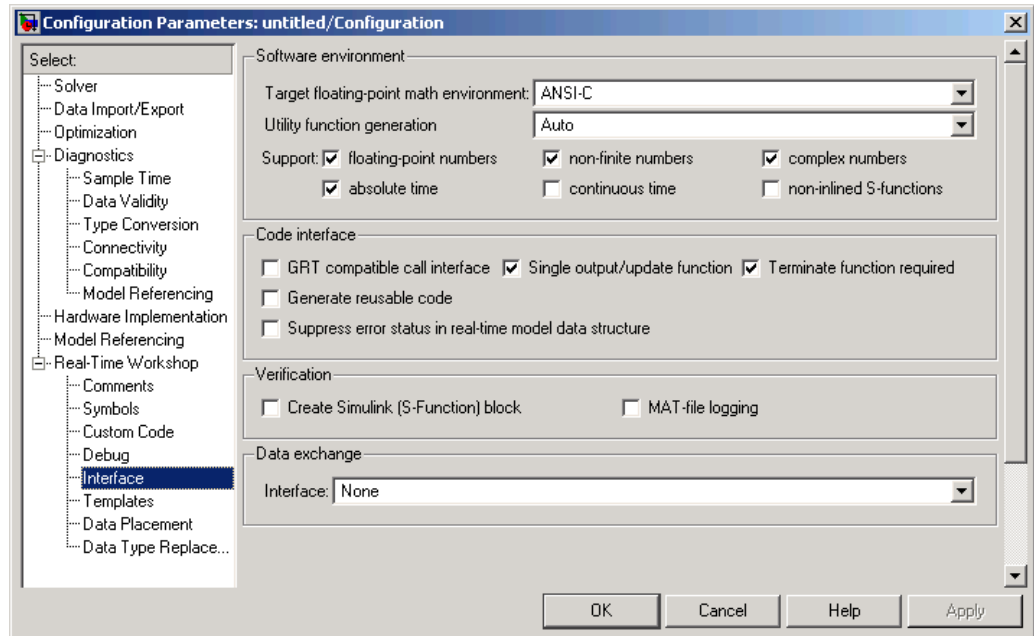
Exceptions to Symbol Formatting Conventions

There are some exceptions to the symbol 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 **Symbol format** 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 **Symbol format** specification does not affect objects (such as signals and parameters) that have a storage class other than Auto (such as ImportedExtern or ExportedGlobal).

Interface Pane

The **Interface** pane provides options software environment, code interface, validation, and data exchange options.



Software Environment Subpane

The **Software Environment** subpane contains options that affect the overall operation of the generated program:

- **Target floating-point math environment:** This menu provides three options:
 - **ANSI_C:** (default) Select this option to generate calls to the ANSI C (ANSI X3.159-1989) math library for floating-point functions.
 - **ISO_C:** Select this option to generate calls to the ISO C (ISO/IEC 9899:1999) math library wherever possible.
 - **GNU:** Select this option to generate calls to the GNU C math library.

If your target compiler supports the ISO C (ISO/IEC 9899:1999) math library, you should select the **ISO_C** option and setting your compiler's ISO C option. This generates calls to the ISO C functions wherever possible (for example, `sqrtf()` instead of `sqrt()` for single precision data) and ensure that you obtain the best performance your target compiler offers.

- **Utility function generation:** See the Real-Time Workshop documentation for information on this option.
- **Support floating-point / non-finite / complex numbers:** These options let you enable or suppress the generation of floating-point, complex, or nonfinite numbers. By default, all three options are selected.

To generate pure integer code, deselect the **Support floating-point numbers** option. If your model requires generation of floating-point data or operations, select the **Support floating-point numbers** option. When **Support floating-point numbers** 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.

The **Support floating-point numbers** option replaces, and inverts the logic of, the **Integer code only** option that was supported in previous releases. Note that for compatibility, models that were configured for **Integer code only** prior to Release 14 are configured automatically with **Support floating-point numbers** deselected, and therefore continue to generate pure integer code.

The **Support non-finite numbers** option is enabled only when **Support floating-point numbers** is selected. This option lets you enable or suppress generation of non-finite values (for example, NaN, Inf).

The **Support complex numbers** option is independent of the other two options. This option lets you enable or suppress generation of complex numbers.

- **Support absolute time:** Certain blocks require the value of either absolute time (that is, the time from the start of program execution to the present time) or elapsed time (for example, the time elapsed between two trigger events). These related options determine how the ERT target provides absolute or elapsed time values to blocks in the model.

By default, **Support absolute time** is selected. In this case, the ERT target generates and maintains integer counters if a block in the model requires absolute or elapsed time values. The target does not generate the

counters if model blocks do not use time values. When **Support absolute time** is deselected, an error is raised at code generation time if any blocks requiring absolute or elapsed time values are present in the model.

For further information on the allocation and operation of absolute and elapsed timers, see the “Timing Services” chapter of the Real-Time Workshop documentation.

- **Support continuous time:** If this option is selected, the ERT target supports code generation for continuous-time blocks. By default, this option is deselected, and the build process generates an error if any continuous-time blocks are present in the model.

Note that continuous time is *not* supported when generating an ERT S-function wrapper (see “Automatic S-Function Wrapper Generation” on page 3-56).

- **Support non-inlined S-functions:** If this option is selected, the ERT target supports code generation for non-inlined S-functions. By default, this option is deselected, and the build process generates an error if any C-MEX S-function that does not have a corresponding TLC implementation (for inlining code generation) is present in the model.

Generation of non-inlined S-functions requires support for both floating-point and non-finite numbers. When the **Support non-inlined S-functions** option is selected, the **Support floating-point numbers** and **Support non-finite numbers** options are automatically selected.

Note that inlining S-functions is highly advantageous in production code generation, for example in implementing device drivers. You may want to deselect **Support non-inlined S-functions** to enforce use of inlined S-functions for code generation.

Code Interface Subpane

The **Code interface** subpane contains options that control whether or not certain model functions are generated and how arguments are passed to functions:

- **GRT compatible call interface:** When this option is selected, Real-Time Workshop Embedded Coder generates model function calls that are compatible with the main program module of the GRT target (`grt_main.c` or `.cpp`). These calls act as wrappers that interface to ERT (Embedded-C format) generated code.

This option provides a quick way to use ERT target features with a GRT-based custom target that has a main program module based on `grt_main.c` or `.cpp`.

Note When **GRT compatible call interface** is selected, **MAT-file logging** must also be selected, and **Suppress error status in real-time model data structure** must be deselected.

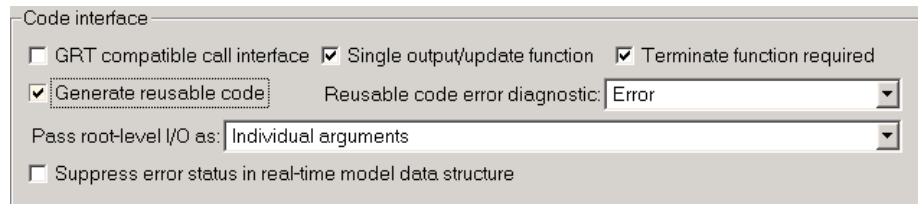
- **Single output/update function:** By default, this option is selected, and the output and update functions are combined in a single `model_step` function. This reduces overhead and allows Real-Time Workshop Embedded Coder to use more local variables in the step function of the model.
- **Terminate function required:** By default, this option is selected, and a `model_terminate` function is generated (for more information, see the description of “`model_terminate`” on page 2-24). Deselect this option if your application is designed to run indefinitely and does not require a terminate function.
- **Generate reusable code:** The **Generate reusable code** option and its related options let you generate reusable, reentrant code from a model or subsystem. When **Generate reusable code** option is deselected (the default), model data structures are statically allocated and accessed directly in the model code. Therefore the model code is neither reusable nor reentrant.

“Model Entry Points” on page 2-22 documents the calling interface generated for the model functions in the default case.

When **Generate reusable code** is selected, the **Code interface** subpane displays and enables the additional options:

- **Reusable code error diagnostic**
- **Pass root-level I/O as**

The figure below shows these options at their default values.



When **Generate reusable code** is selected, data structures such as block states, parameters, external outputs, and so on, are passed in (by reference) as arguments to `model_step` and other generated model functions. These data structures are also exported with `model.h`.

The **Pass root-level I/O as:** menu provides options that control how model inputs and outputs at the root level of the model are passed in to the `model_step` function. The options are

- **Individual arguments:** This option is the default. Each root-level model input and output is passed to `model_step` as a separate argument.
- **Structure reference:** When this option is selected, all root-level inputs are packed into a struct that is passed to `model_step` as an argument. Likewise, all root-level outputs are packed into a struct that is also passed to `model_step` as an argument.

In some cases, selecting **Generate reusable code** may 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, the **Reusable code error diagnostic** menu is enabled when **Generate reusable code** is selected. This menu offers a choice of three severity levels for diagnostics to be displayed in such cases:

- **None:** Build proceeds without displaying a diagnostic message.
- **Warning:** Build proceeds after displaying a warning message.

- Error: Build aborts after displaying an error message.

In some cases, Real-Time Workshop Embedded Coder 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.

- **Suppress error status in real-time model data structure:** If you do not need to log or monitor error status in your application, select this option.

By default, the real-time model data structure (`rtModel`) includes an error status field. This field lets you log and monitor error messages with macros provided for this purpose (see “`rtModel` Accessor Macros” on page 2-3). If **Suppress error status in real-time model data structure** is selected, the error status field is not included in `rtModel`. Selecting this option may also cause the real-time model data structure to disappear completely from the generated code.

When generating code for multiple models that are integrated together, make sure that the **Suppress error status in real-time model data structure** option is set the same for all of the models. Otherwise, the integrated application may exhibit unexpected behavior. For example, if the option is selected in one model but not in another, the error status may or may not be registered by the integrated application.

Do not select **Suppress error status in real-time model data structure** if the **MAT-file logging** option is also selected. The two options are incompatible.

Verification Subpane

The **Verification** subpane contains options that are useful for verifying generated code in Simulink.

- **Create Simulink (S-Function) block:** Selecting this option lets you generate an S-function wrapper that calls your C or C++ code from within Simulink. S-function wrappers provide a standard interface between Simulink and externally written code, allowing you to integrate your code into a model with minimal modification. See “Automatic S-Function Wrapper Generation” on page 3-56 for information on this feature.
- **MAT-file logging:** This option enables or suppresses MAT-file logging. By default, **MAT-file logging** is deselected. This default is appropriate for embedded applications, which typically do not support a file system. Also, suppression of MAT-file logging 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 has the following effects:
 - Under certain conditions, code and storage associated with root output ports are eliminated, achieving further efficiency. See “Use the Virtualized Output Ports Optimization” on page 3-48 for information.
 - The `model_step` function does not check the current time against the stop time. Therefore the generated program runs indefinitely, regardless of the setting of the model’s stop time. The `ert_main` program displays a message notifying you that the program runs indefinitely.

MAT-file logging requires support for both floating-point and non-finite numbers. When the **MAT-file logging** option is selected, the **Support floating-point numbers** and **Support non-finite numbers** options are automatically selected.

- **MAT-file variable name modifier:** This menu is displayed when **MAT-file logging** is selected. The menu selects a string to be added to the variable names used when logging data to MAT-files.

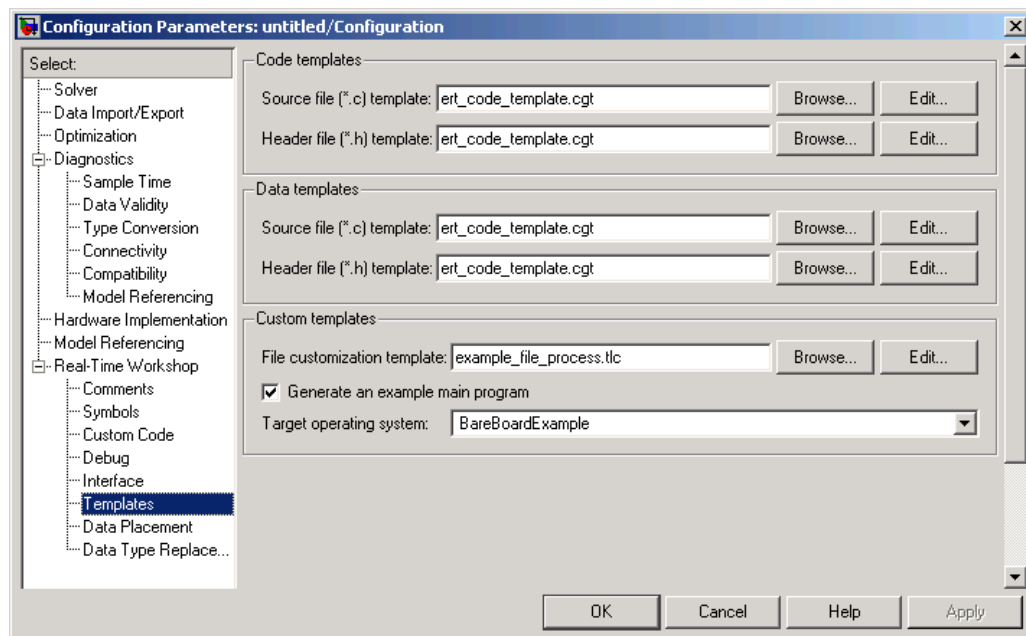
Data Exchange Subpane

The **Data Exchange** subpane contains options related to interfacing model data to systems external to the generated code. These options are selected with the **Interface** menu. Depending on the choice selected from the **Interface** menu, different suboptions are displayed dynamically below the **Interface** menu. The **Interface** menu offers the following choices:

- **C-API**: Generate C API code that allows externally written code to access block outputs (signals) and/or parameters. For documentation of the C API for signals and parameters, see the Real-Time Workshop documentation.
- **External mode**: Generate external mode support code. If you want to deploy external mode code on an embedded target, see “Using External Mode with the ERT Target” on page 3-51 for special considerations.
- **ASAP2**: Export an ASAP2 file containing information about the model during the code generation process. See the “Generating an ASAP2 File” section of the Real-Time Workshop documentation for detailed information.
- **None**: (default) No data exchange code is generated.

Templates Pane

The **Templates** pane contains advanced options that enable you to customize generated code.



Code Templates and Data Templates Subpanes

Code and data templates provide extensive code customization features, which are described in the Module Packaging Features document. Brief descriptions of the options for specifying such templates follow:

- **Source file (*.c) template:** Use this option to create or edit a code template.
- **Course file (*.h) template:** Use this option to create or edit a data template.

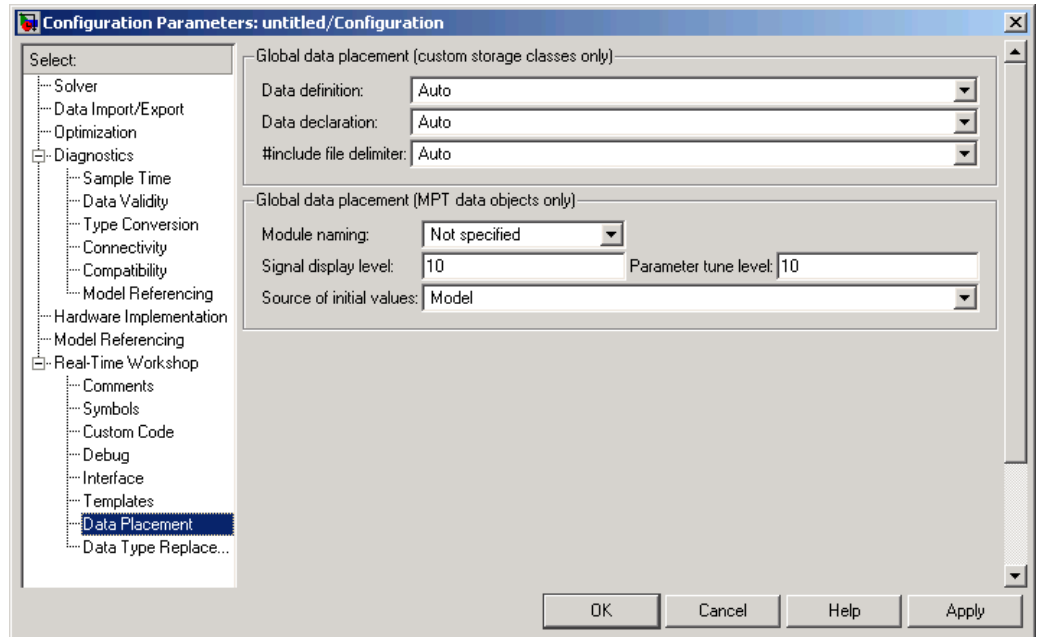
See also “Generating Custom File Banners” on page 5-47 for a simple example of how a code template can be applied to generate customized comment sections in generated code files.

Custom Templates Subpane

- **File customization template:** This option lets you specify a *custom file processing template* (CFP) template file. CFP templates let you customize generated code by organizing generated code into sections (such as includes, typedefs, functions, and more). A CFP template can emit code, directives, or comments into each section as required. See “Custom File Processing” on page 5-26 for detailed information.
- **Generate an example main program:** This option and the related **Target operating system** menu let you generate a model-specific example main program module. See “Generating the Main Program” on page 2-8.

Data Placement Pane

The **Data Placement** pane contains advanced options for Module Packaging Features. For details on using these options, see the Module Packaging Features document.

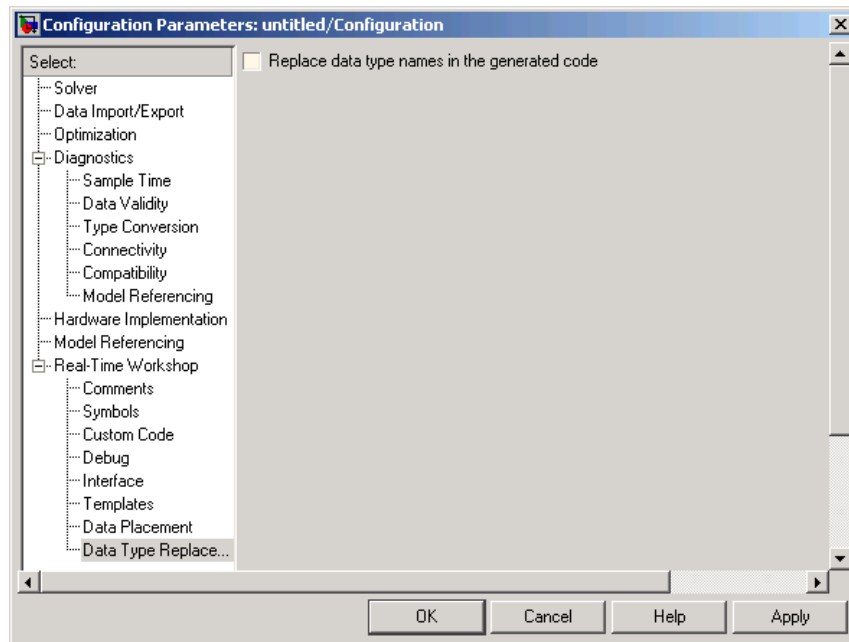


- **Data definition:** Use this option to specify whether data is to be defined in the generated source file or in a single separate header file.
- **Data declaration:** Use this option to specify whether data is to be declared in the generated source file or in a single separate header file.
- **#include file delimiter:** Use this option to specify the #include file delimiter to be used in generated files that contain the #include preprocessor directive for MPF data objects.
- **Module naming:** This option lets you name the generated module using the same name as the model or a user-specified name.
- **Signal display level:** Use this option to control whether signal data objects are to be declared as global data in the generated code.

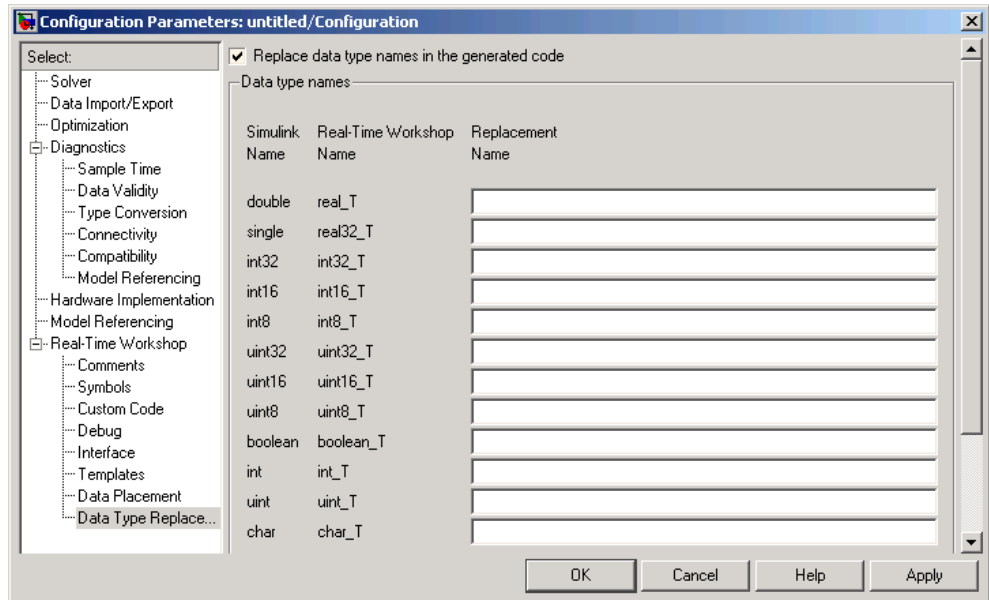
- **Parameter tune level:** This option lets you declare a parameter data object as tunable global data in the generated code.
- **Source of initial values:** Use this option to specify the source that initializes the model's signals in the generated code.

Data Type Replacement Pane

The **Data Type Replacement** pane provides an interface for replacing built-in data type names with user-defined replacement data type names in the generated code for your model.



If you select **Replace data type names in the generated code**, the **Data type names** table is displayed:



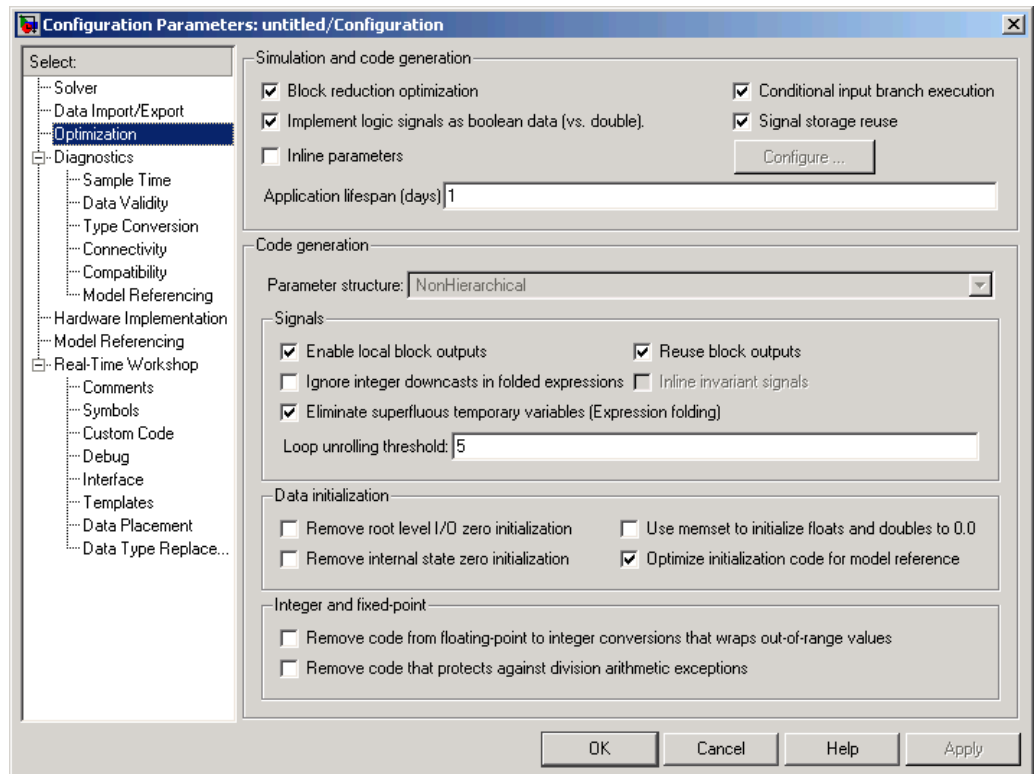
The table **Data type names** lists each Simulink built-in data type name along with its Real-Time Workshop data type name. Selectively fill in fields in the third column with your replacement data types. Each replacement data type should be the name of a `Simulink.AliasType` object that exists in the base workspace. Replacements may be specified or not for each individual built-in type.

For each replacement data type entered, the `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces. For `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, 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. 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.

Optimization Pane

Most of the options in the **Optimization** pane are common to all targets, including all options listed in the **Simulation and code generation** and **Signals** subpanes. These are described in the Real-Time Workshop documentation.

When you select the ERT target (or a target derived from the ERT target), additional options are displayed. These options are described below.



Code Generation Subpane

The **Parameter structure** menu lets you control how parameter data is generated for reusable subsystems. (If you are not familiar with reusable subsystem code generation, see “Nonvirtual Subsystem Code Generation Options” in the Real-Time Workshop documentation.)

The **Parameter structure** menu is enabled when the **Inline parameters** option is on. The menu lets you select the following options:

- Hierarchical: This option is the default. When the Hierarchical option is selected, Real-Time Workshop Embedded Coder generates a separate header file, defining an independent parameter structure, for each subsystem that meets the following conditions:
 - The Reusable function option is selected in the subsystem’s **RTW system code** menu, and the subsystem meets all conditions for generation of reusable subsystem code.
 - The subsystem does not access any parameters other than its own (such as parameters of the root-level model).

When the Hierarchical option is selected, each generated subsystem parameter structure is referenced as a substructure of the root-level parameter data structure, which is therefore called a hierarchical data structure.

- Non-hierarchical: When this option is selected, Real-Time Workshop Embedded Coder generates a single parameter data structure. This is a flat data structure; subsystem parameters are defined as fields within the structure. Using a nonhierarchical data structure can reduce compiler padding between word boundaries in many cases; this produces more efficient compiled code.

Data Initialization Subpane

- **Remove root-level I/O zero initialization:** When this option is off (the default), initialization code for root-level inports and outports whose value is zero is generated. Otherwise, initialization code for such inports and outports is not generated.
- **Use memset to initialize floats and doubles to 0.0:** When **Use memset to initialize floats and doubles to 0.0** is off (the default), additional code is generated to set float and double storage explicitly to the value 0.0. When this option is on, the memset 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 turning on this option.

- **Remove internal state zero initialization:** When this option is off (the default), initialization code that initializes internal work structures (for example, block states and block outputs) to zero is generated. Otherwise, the initialization code is not generated.
- **Optimize initialization code for model reference:** You should deselect this option if your model meets both the following conditions:
 - The model contains an enabled subsystem
 - The model is referred to from another model with a Model block.Otherwise, you should leave the option selected (the default).

Integer and Fixed-Point Subpane

- **Remove code from floating-point to integer conversions that wraps out-of-range values:** This option causes Real-Time Workshop to remove code that ensures that execution of the generated code produces the same results as simulation when out-of-range conversions occur. 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.

Enabling this option affects code generation results only for out-of-range values and cannot cause code generation results to differ from simulation results for in-range values.

- **Remove code that protects against division arithmetic exceptions:** This option suppresses generation of code that guards against fixed-point division by zero. By default, this option is deselected.

When you select this option, simulation results and results from generated code may no longer be in bit-for-bit agreement.

Simulation and Code Generation Subpane

Note The **Application lifespan (days)** parameter in the **Simulation and code generation** subpane lets you minimize the allocation of memory for absolute and elapsed time counters. The word size of the counters (8, 16, 32, or 64 bits), is allocated optimally, to accommodate the maximum value specified in **Application lifespan (days)** field. For further information on the allocation and operation of absolute and elapsed timers, see the “Timing Services” section of the Real-Time Workshop documentation.

Tips for Optimizing the Generated Code

Real-Time Workshop Embedded Coder 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.

Use Auto-Optimized Targets

To make it easier for you to generate the most efficient code for your target CPU, Real-Time Workshop Embedded Coder 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 5-18.

Use Configuration Wizard Blocks

Real-Time Workshop Embedded Coder 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 5-53 for detailed information.

Set 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. See the “Hardware Implementation Pane” section of the Simulink User’s Guide and the “Hardware Implementation Options” section of 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.

Remove Unnecessary Initialization Code

Consider selecting the **Remove internal state zero initialization** and **Remove root-level I/O zero initialization data** options on the **Data initialization** subpane under **Optimization**.

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 state zero initialization**, it is not guaranteed that memory is in a known state each time the generated code begins execution. If you turn the option on, running a model (or a generated S-function) multiple times can result in different answers for each run.

This behavior is sometimes desirable. For example, you can turn on **Remove internal state zero initialization** if you want to test the behavior of your design during a warm boot (that is, a restart without full system reinitialization).

In cases where you have turned on **Remove internal state zero initialization** but still want to get the same answer on every run from a Real-Time Workshop Embedded Coder generated S-function, 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 “Data Initialization Subpane” on page 3-42.

Note that the code still initializes data structures whose value is not zero when **Remove internal state zero initialization** and **Remove root-level I/O zero initialization data** are selected.

Note also that data of `ImportedExtern` or `ImportedExternPointer` storage classes is never initialized, regardless of the settings of these options.

Generate 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.

Disable MAT-File Logging

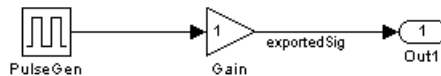
Clear the **MAT-file logging** option in the **Verification** section of the **Interface** pane. This setting is the default, and is recommended for embedded applications because it eliminates the extra code and memory usage required to initialize, update, and clean up logging variables. In addition to these efficiencies, clearing the **MAT-file logging** option lets you exploit further efficiencies under certain conditions. See “Use the Virtualized Output Ports Optimization” on page 3-48 for information.

Note also that code generated to support MAT-file logging invokes `malloc`, which may be undesirable for your application.

Use the 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.

To illustrate this feature, consider the model shown in the following block diagram. Assume that the signal exportedSig has exportedGlobal storage class.



In the default case, the output of the Gain block is written to the signal storage location, exportedSig. No code or data is generated for the Out1 block, which has become, in effect, a virtual block. This is shown in the following code fragment.

```

/* Gain Block: <Root>/Gain */
exportedSig = rtb_PulseGen * VirtOutPortLogOFF_P.Gain_Gain;
  
```

In cases where either the **MAT-file logging** option is enabled, or FullRootOutputVector = 1, the generated code represents root output ports as members of an external outputs vector.

The following code fragment was generated from the same model shown in the previous example, but with **MAT-file logging** enabled. The output port is represented as a member of the external outputs vector VirtOutPortLogON_Y. The Gain block output value is copied to both exportedSig and to the external outputs vector.

```

/* Gain Block: <Root>/Gain */
exportedSig = rtb_PulseGen * VirtOutPortLogON_P.Gain_Gain;

/* Outputport Block: <Root>/Out1 */
VirtOutPortLogON_Y.Out1 = exportedSig;
  
```

The overhead incurred by maintenance of data in the external outputs vector can be significant for smaller models being used to perform benchmarks.

Note that you can force root output ports to be stored in the external outputs vector (regardless of the setting of **MAT-file logging**) by setting the TLC variable `FullRootOutputVector` to 1. You can do this by adding the statement

```
%assign FullRootOutputVector = 1
```

to the 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.

For more information on how to control signal storage in generated code, see the “Working with Data Structures” section of the Real-Time Workshop documentation.

Use Stack Space Allocation Options

Real-Time Workshop 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 Structures” 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. 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 (Simulink) and target systems. Real-Time Workshop Embedded Coder 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 figure below shows the **Interface** pane of the Configuration Parameters dialog, 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 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

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

Generating a Code Generation Report

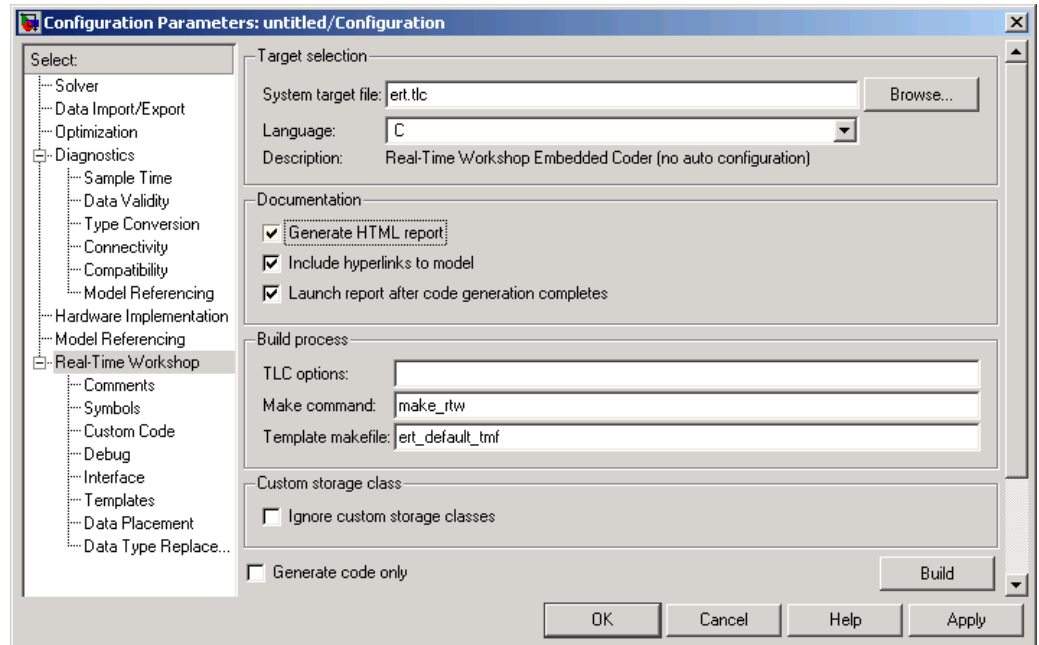
The Real-Time Workshop Embedded Coder code generation report is an enhanced version of the HTML code generation report normally generated by Real-Time Workshop. The report consists of several sections:

- The **Generated Source Files** section of the Contents pane contains a table of source code files generated from your model. You can view the source code in a MATLAB Web browser window. Optional 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.
- The **Summary** section lists version and date information, TLC options used in code generation, and Simulink model settings.
- The **Optimizations** section lists the optimizations used during the build, and also those that are available. If you chose options that led to generation of nonoptimal code, they are marked in red. This section can help you select options that better optimize your code.
- The report also includes information on other code generation options, code dependencies, and links to relevant documentation.

To generate a code generation report,

- 1** Open the Configuration Parameters dialog and select the **Real-Time Workshop** pane.
- 2** In the **Documentation** subpane, select **Generate HTML report**. By default, **Include hyperlinks to model** and **Launch report after code generation completes** are also selected, as shown in the figure below.

You can deselect either or both these options if desired.



- 3 Follow the usual procedure for generating code from your model or subsystem.
- 4 Real-Time Workshop 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 after code generation completes**, Real-Time Workshop automatically opens a MATLAB Web browser window and displays the code generation report.

If you did not select **Launch report after code generation completes**, 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 **Include hyperlinks to model**, hyperlinks to blocks in the generating model are created in the report files. When you view the report files in MATLAB, clicking on these hyperlinks displays and highlights the referenced blocks in the model.

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 **Include hyperlinks to model** check box. 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 the Real-Time Workshop documentation for details.
-

Automatic S-Function Wrapper Generation

An S-function wrapper is an S-function that calls your C or C++ code from within Simulink. S-function wrappers provide a standard interface between Simulink and externally written code, allowing you to integrate your code into a model with minimal modification. 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

This is useful for software-in-the-loop (SIL) code verification, as well as for simulation acceleration purposes.

When the **Create Simulink (S-Function) block** option is on, Real-Time Workshop generates an additional source code file, *model_sf.c* or *.cpp*, in the build directory. This module contains the S-function that calls the Real-Time Workshop Embedded Coder code that you deploy. You can use this S-function within Simulink.

The build process then compiles and links *model_sf.c* or *.cpp* with *model.c* or *.cpp* and the other Real-Time Workshop Embedded Coder generated code modules, building a MEX-file. The MEX-file is named *model_sf.mexext*. (*mexext* is the file extension for MEX-files on your platform, as given by the MATLAB *mexext* command.) The MEX-file is stored in your working directory. Finally, Real-Time Workshop 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, select **Use Embedded Coder** and click **Build**.

Limitations

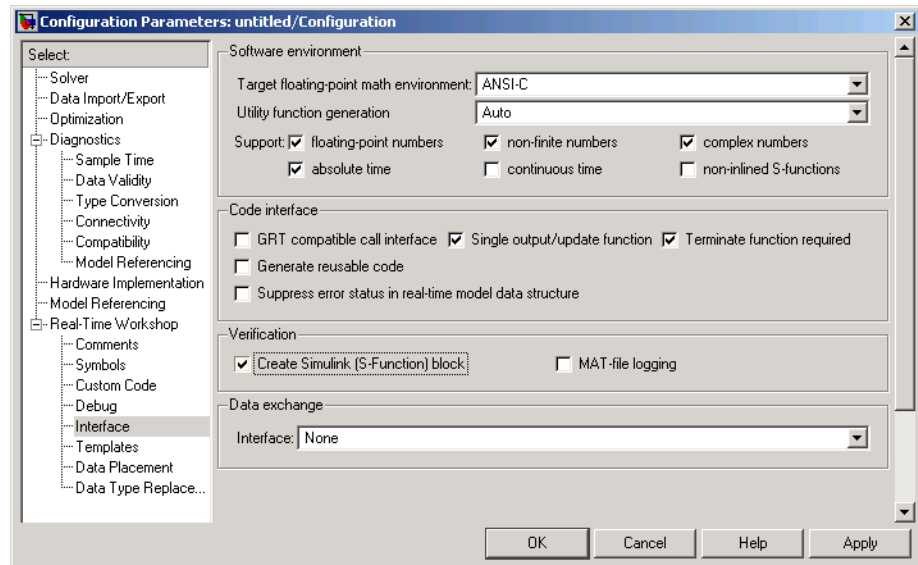
The following limitations apply to Real-Time Workshop Embedded Coder S-function wrapper generation:

- Continuous sample time is not supported. The **Support continuous time** option should not be selected when generating an 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 an Embedded Coder S-function wrapper.
- You cannot use multiple instances of an Embedded Coder generated S-function block within a model, because the code uses static memory allocation. Each instance potentially can overwrite global data values of the others.
- Embedded Coder generated 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.

Generating an S-Function Wrapper

To generate an S-function wrapper for your Real-Time Workshop Embedded Coder code:

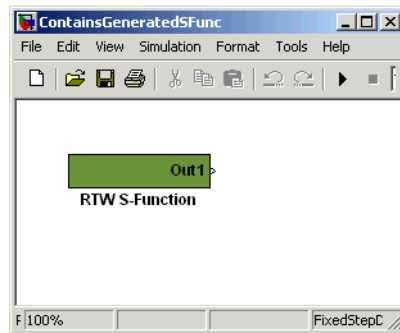
- 1 Open the Configuration Parameters dialog.
- 2 Select the **Interface** pane.
- 3 Select the **Create Simulink (S-Function) block** option, as shown in this figure.



- 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 state zero initialization**
 - **Use memset to initialize floats and doubles to 0.0**

(See also “Data Initialization Subpane” on page 3-42.)

- 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 blocks or models in Simulink.

Custom Storage Classes

Introduction to Custom Storage Classes (p. 4-3)

Overview of how Real-Time Workshop Embedded Coder's custom storage classes (CSCs) extend your control over the representation of data in an embedded algorithm.

Custom Storage Classes and Simulink Data Objects (p. 4-5)

Relationship between custom storage classes and Simulink data class packages and objects; predefined custom storage classes for signal and parameter objects; how to set custom storage class properties of data objects for use in code generation; code generation example using signal objects with custom storage classes.

Designing Custom Storage Classes (p. 4-15)

Using the Custom Storage Class Designer to implement your own custom storage classes.

Creating Packages with CSC Definitions (p. 4-30)

Using the Simulink Data Class Designer to create data object packages associated with custom storage classes.

Defining Advanced Custom Storage Class Types (p. 4-34)

Defining custom storage classes from scratch, including associated TLC code generation implementation; using advanced mode of the Custom Storage Class Designer.

GetSet Custom Storage Class for Data Store Memory (p. 4-38)	A special storage class for use with Data Store Memory blocks.
Requirements and Restrictions for Use of CSCs (p. 4-41)	Setting related options correctly for code generation with CSCs; general restrictions; restrictions that apply to the use of CSC in models that use the Model Referencing feature.
Older Custom Storage Classes (Prior to Release 14) (p. 4-43)	Compatibility information on custom storage classes provided in versions of Real-Time Workshop Embedded Coder prior to Version 4.0 (MATLAB Release 14).

Introduction to Custom Storage Classes

In Real-Time Workshop, 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 Real-Time Workshop, the term “storage class” is not synonymous with the term “storage class specifier”, as used in the C language.

Real-Time Workshop 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 Structures” 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’s *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 Real-Time Workshop. Real-Time Workshop Embedded Coder provides

- 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, you should read the “Simulink Data Objects and Code Generation” section of the Real-Time Workshop documentation.

- The Custom Storage Class Designer (`cscdesigner`) tool. 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.
- The Simulink Data Class Designer. This chapter describes how to use the Simulink Data Class Designer to create a data object package and associate your own custom CSC definitions with classes contained in the package. For a general description of the Simulink Data Class Designer, see the Simulink documentation.

Custom Storage Classes and Simulink Data Objects

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” on page 4-15.

This section discusses the following topics related to predefined CSCs and their use in code generation:

- “Predefined CSCs” on page 4-6 discusses the ready-to-use CSCs provided for parameter and signal objects.
- “Setting the Custom Storage Class Properties” on page 4-9 demonstrates how to configure the CSC-related properties of parameter and signal objects.
- “Generating Code with CSCs” on page 4-10 guides you through the steps required to generate code using CSCs, using signal objects as an example.

Predefined CSCs

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

The `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`. Summary of Predefined Simulink CSCs for Signal and Parameter Objects on page 4-7 lists the predefined set of CSCs provided by Real-Time Workshop Embedded Coder.
- `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. Summary of Instance-Specific Properties for CSCs on page 4-8 lists instance-specific properties for the predefined set of CSCs provided by Real-Time Workshop Embedded Coder.

Note that 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. Summary of Predefined Simulink CSCs for Signal and Parameter Objects on page 4-7 indicates whether each class is valid for parameter or signal objects.

Summary of Predefined Simulink CSCs for Signal and Parameter Objects

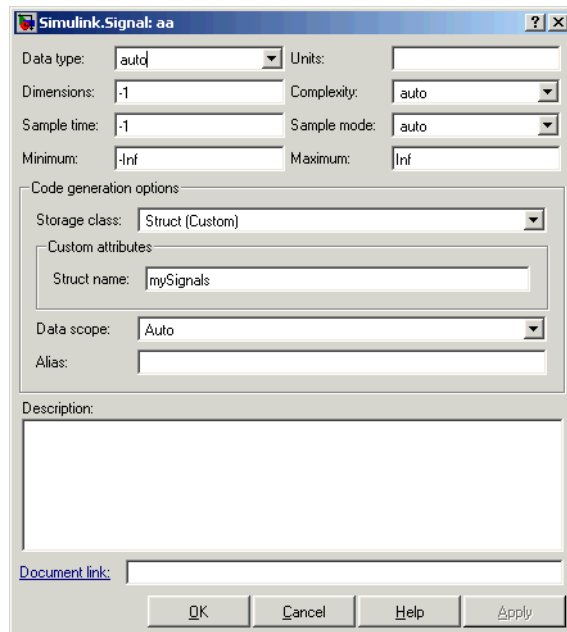
Class Name (Enumerated)	Available for Signals	Available for Parameters	Purpose
BitField	Y	Y	Generate a struct declaration that embeds Boolean data in named bit fields.
Const	N	Y	Generate a constant declaration with the const type qualifier.
ConstVolatile	N	Y	Generate declaration of volatile constant 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.

Summary of 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.

Setting the 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. This dialog appears in the right pane of the Model Explorer. Alternatively, you can launch the dialog independently by right-clicking the relevant object in center pane of the Model Explorer. 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:

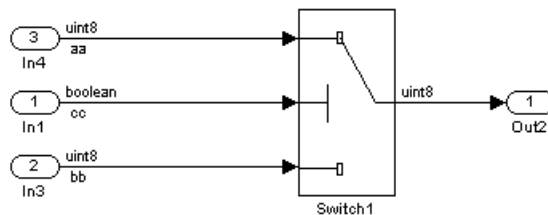
```
aa = Simulink.Signal;  
aa.RTWInfo.StorageClass = 'Custom';  
aa.RTWInfo.CustomStorageClass = 'Struct';  
aa.RTWInfo.CustomAttributes.StructName = 'mySignals';
```

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 the next section, “Generating Code with CSCs” on page 4-10.

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 “Requirements and Restrictions for Use of CSCs” on page 4-41 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

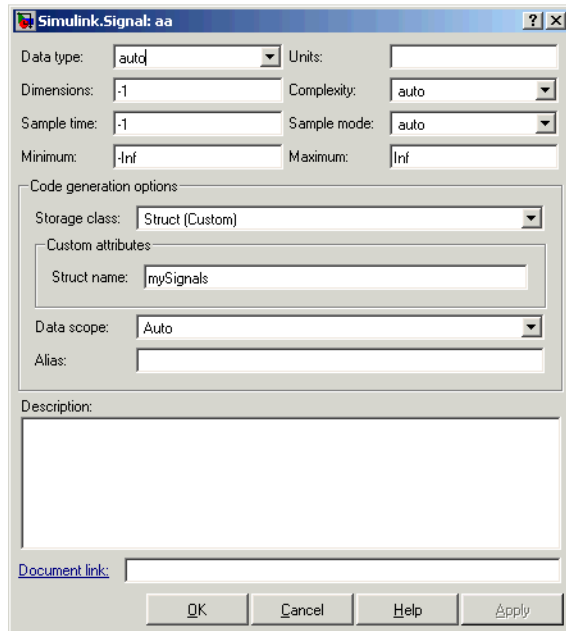
The next step is to instantiate signal objects. You can do this with MATLAB commands as shown below.

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

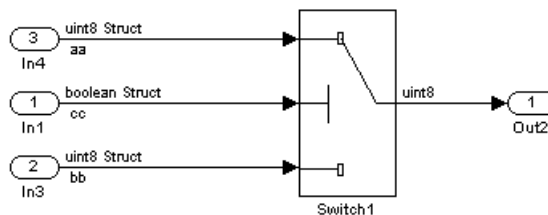
Alternatively, you can create the signal objects in the Simulink Model Explorer by clicking **Add Simulink Signal** or selecting **Add Simulink.Signal** from the **Add** menu.

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 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 **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.

Designing Custom Storage Classes

The Custom Storage Class Designer (`cscdesigner`) is a tool for creating and managing CSCs. The Custom Storage Class Designer lets you

- Load existing CSCs and view and edit their definitions
- Create new CSCs, or copy and modify existing CSC definitions
- Control placement of data objects in memory (for example, in RAM, ROM, and flash memory sections)
- Preview pseudocode generated from CSC definitions
- Verify the correctness and consistency of CSC definitions
- Save CSC definitions

Custom Storage Class Designer Overview

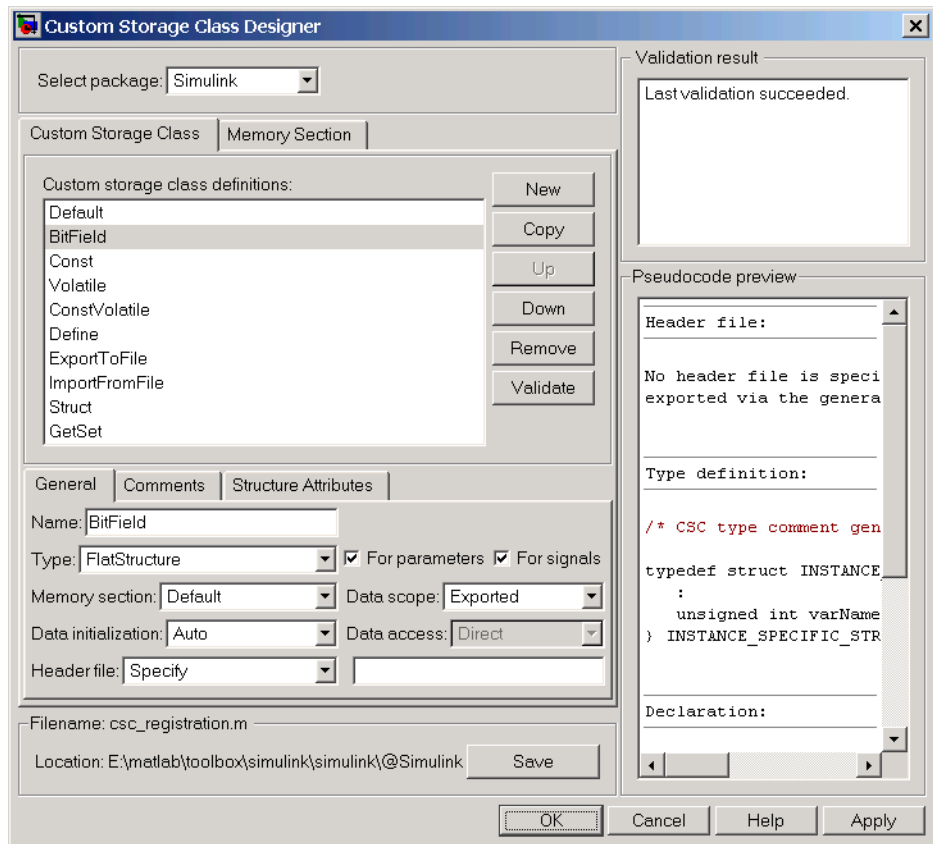
This section provides a quick introduction to the Custom Storage Class Designer, with references to the detailed descriptions located in the following section, “Using the Custom Storage Class Designer” on page 4-17.

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

```
cscdesigner
```

When first opened, `cscdesigner` scans all data class packages on the MATLAB path to detect packages that have a CSC registration file. A message window is displayed while scanning proceeds.

When the scan is complete, the Custom Storage Class Designer window opens (see Custom Storage Class Designer Window on page 4-16).



Custom Storage Class Designer Window

The window is divided into several panels:

- **Select package** panel: 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 4-17 for details.
- **Custom Storage Class / Memory Section** properties panel: 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 Maintaining CSC and Memory Section Definitions” on page 4-18.

When the **Custom Storage Class** tab is selected, you can select a CSC from a list, and edit its properties. See “Editing Properties of CSCs” on page 4-19 for details.

When the **Memory Section** tab is selected, you can select a *memory section* definition from a list, and edit its properties. Each CSC has an associated memory section definition. A memory section definition is a named collection of properties related to placement of an object in memory. The 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 4-26 for details.

- **Filename** panel: 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 4-29 for details.
- **Pseudocode preview** panel: 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 4-28 for details.
- **Validation result** panel: Displays any errors encountered when the currently selected CSC definition is validated. See “Validating CSC Definitions” on page 4-29 for details.

Using the Custom Storage Class Designer

This section provides a detailed description of the Custom Storage Class Designer window and each of its functions.

Selecting a Data Class Package

A CSC definition is uniquely associated with a Simulink data class package. The link between a CSC and a 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.

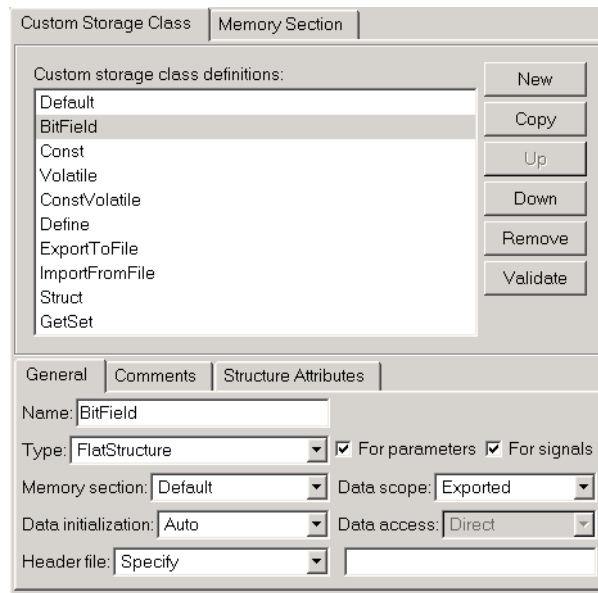


The **Select package** panel contains a menu of names of all detected data class packages that have a CSC registration file. At least one such package, the Simulink package, is always present.

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 Maintaining CSC and Memory Section Definitions

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



The list at the top of the panel displays the definitions for the currently selected package. To select a definition for viewing and editing, click on the desired list entry.

The properties of the selected definition are displayed in the area below the list. The number and type of properties vary for different types of CSC and memory section definitions. See “Editing Properties of CSCs” on page 4-19 for specific information about the properties of the predefined CSCs. See “Editing Memory Section Definitions” on page 4-26 for specific information about the properties of the predefined memory section definitions.

The buttons to the right of the list perform the following functions:

- **New:** Creates a new CSC definition with default values.
- **Copy:** Creates a copy of the selected definition. Copies are given a default name by 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.

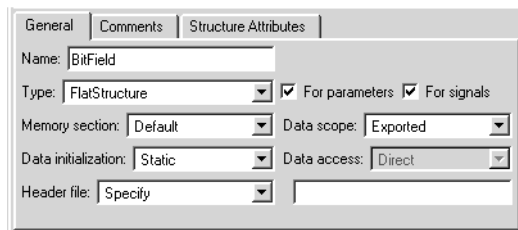
Editing Properties of CSCs

To view and edit the properties of a CSC, click on the **Custom Storage Class** tab in the **Custom Storage Class / Memory Section** panel. Then, select a CSC from the **Custom storage class** definition list. The CSC properties are divided into several categories, selected by tabs. The number and type of tabs and properties depend on the selected class and in some cases on the property values themselves.

As you change property values, the effect upon 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 categories and corresponding tabs are as follows:

General. Properties in the **General** category are common to all CSCs. These properties are shown in the following figure.



Properties in the **General** category are:

- **Name:** Class name (displayed in **Custom storage class** list).
- **Type:** If Unstructured is selected, objects of this class generate unstructured storage declarations (for example, scalar or array variables), for example:

```
datatype dataname[dimension];
```

If FlatStructure is selected, 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 4-23).

Note Certain data layouts (for example, nested structures) 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 4-34 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:** This menu selects one of the memory section definitions defined in the **Memory Section** panel. See “Editing Memory Section Definitions” on page 4-26.
- **Data scope:** Controls the scope of symbols generated for data objects of this CSC.
 - Auto: Symbol scope is determined internally by Real-Time Workshop. If possible, symbols have File scope. Otherwise, they have Exported scope.
 - Exported: Symbol is exported to external code with either *model.h* (default), or by the header file specified by the **Header File** field (see below).
 - Imported: Symbol is imported from external code with the header file specified under the **Header File** tab (see below). If you do not specify a header file, an extern directive is generated in *model_private.h*.

Note that for imported data if the **Data initialization** field (see below) specifies the Macro option, a header file *must* be specified.

- File: Symbol has scope within the file that defines it. File scope requires that the symbol is 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 individual data objects.
- **Data initialization:** This field controls how storage is initialized in generated code. Select one of
 - 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. In other words, it is not available when the class is configured for generation of structured data, or for signals.

- Instance specific: Specify one of the above initialization options when configuring each instance of the object.
- **Data access:** This field is enabled when **Data scope** is set to Imported. This field controls whether or not an imported symbol is declared as a pointer. Select either
 - Direct: Symbol is declared as a simple variable, such as

```
extern myType myVariable;
```

- Pointer: Symbol is declared as a pointer variable, such as

```
extern myType *myVariable;
```

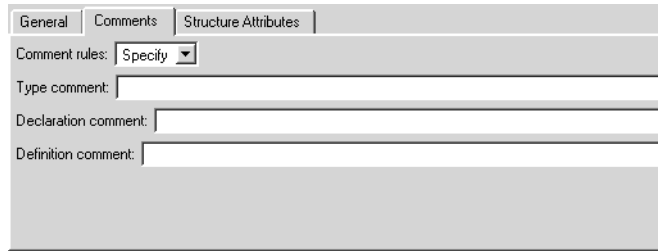
- **Header file:** If Specify is selected, an edit field is displayed to the right of this property. This lets you specify a default header file for exported or imported storage declarations. Specify the full filename, including the filename extension (such as .h) and the desired quote or bracket delimiters.

If Instance specific is selected, objects of this class have the `RTWInfo.CustomAttributes.HeaderFile` property. This property allows you to define the name of the header file that contains exported or imported variable declarations and definitions for each object of the class.

The **Header File** specification interacts with the **Data scope** and **Data initialization** properties as follows:

- If the **Data scope** of the class is set to Imported, and **Data initialization** is set to Macro, you must specify a header file. A `#include` directive for the header file is generated.
- If the **Data scope** of the class is set to Exported, specifying a header file is optional. If you specify a header file, the custom storage class generates a header file containing the storage declarations to be exported. Otherwise, the storage declarations are exported with `model.h`.

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

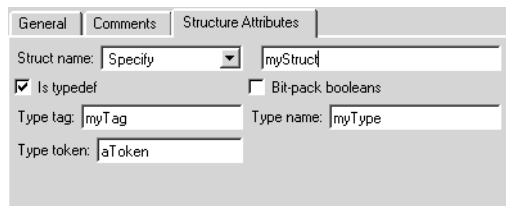


Comments must conform to the ANSI C standard (`/* ... */`). Use `\n` to specify a new line.

Properties in the **Comments** 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 Real-Time Workshop.
- **Type comment:** The comment entered in this field precedes the typedef or struct definition for structured data.
- **Declaration comment:** Comment that precedes the storage declaration.
- **Definition comment:** Comment that precedes the storage definition.

Structure Attributes. The **Structure Attributes** panel gives you detailed control over code generation for structs (including bitfields). The **Structure Attributes** tab is displayed for CSCs whose **Type** parameter is set to `FlatStructure`. The following figure shows the **Structure Attributes** panel.



Structure Attributes Panel

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

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

If Specify is selected, an edit field is displayed (as shown in Structure Attributes Panel on page 4-23) 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 4-23.

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:

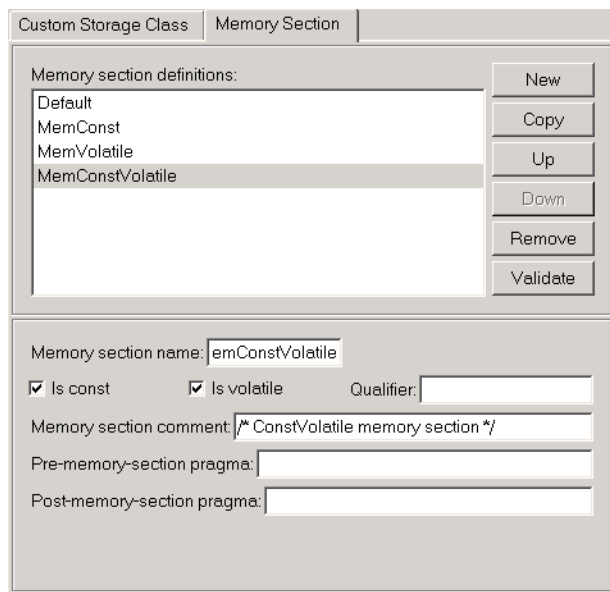
```
/* CSC declaration comment generated by default */  
  
extern myType MyStruct;
```

Definition:

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

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 **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:

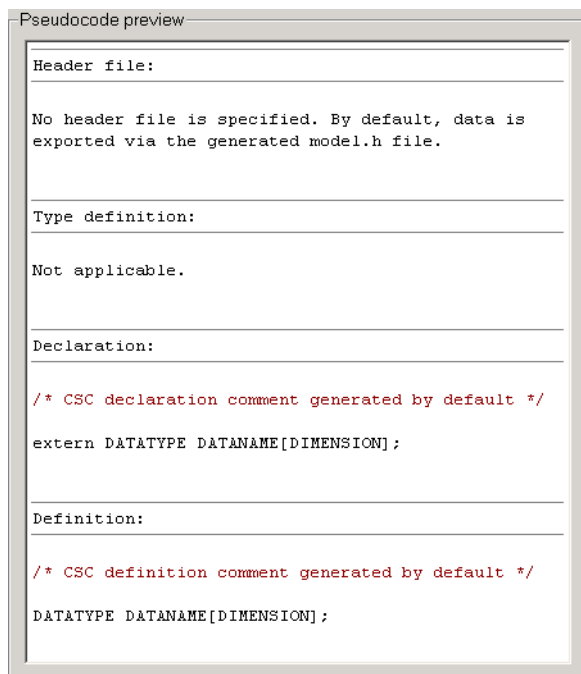
- **Memory section name:** Name of the memory section (displayed in **Memory section definitions** list).
- **Is const:** If selected, a `const` qualifier is added to the symbol declarations.
- **Is volatile:** If selected, a `volatile` qualifier is added to the symbol declarations.
- **Qualifier:** 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.

- **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

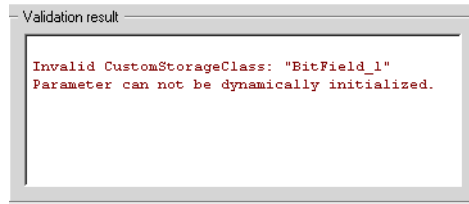
If you click **Validate** on the **Memory Section** panel, the **Pseudocode preview** panel displays a preview of code that is generated from objects of the given class. The panel also displays messages (in blue) to highlight changes as they are made. The code preview changes dynamically as you edit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



```
Pseudocode preview
Header file:
No header file is specified. By default, data is
exported via the generated model.h file.
Type definition:
Not applicable.
Declaration:
/* CSC declaration comment generated by default */
extern DATATYPE DATANAME[DIMENSION];
Definition:
/* CSC definition comment generated by default */
DATATYPE DATANAME[DIMENSION];
```

Validating CSC Definitions

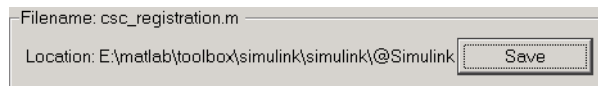
To validate a CSC definition, select the definition and click **Validate** on the **Memory Section** panel. The Custom Storage Class Designer then checks the definition for consistency. The **Validation result** panel displays any errors encountered when a selected CSC definition is validated.



Validation is also performed whenever CSC definitions are saved. In this case, all CSC definitions are selected. (See “Saving Your Definitions” on page 4-29.)

Saving Your Definitions

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



If errors occur, they are reported in the **Validation result** panel. The definitions are still saved, however. You should correct all validation errors and resave your definitions.

Note If you edit a CSC definition that has been assigned to existing parameter or signal objects, you must restart MATLAB after editing and saving the CSC definition.

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.

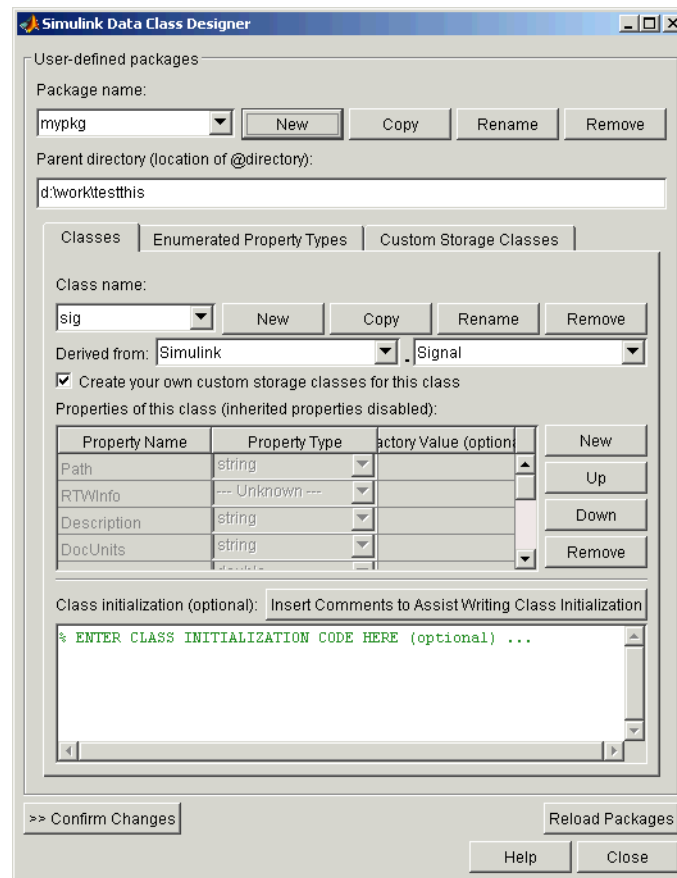
- 1 Open the Simulink Data Class Designer by typing the following command at the MATLAB command prompt:

```
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.
- 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.

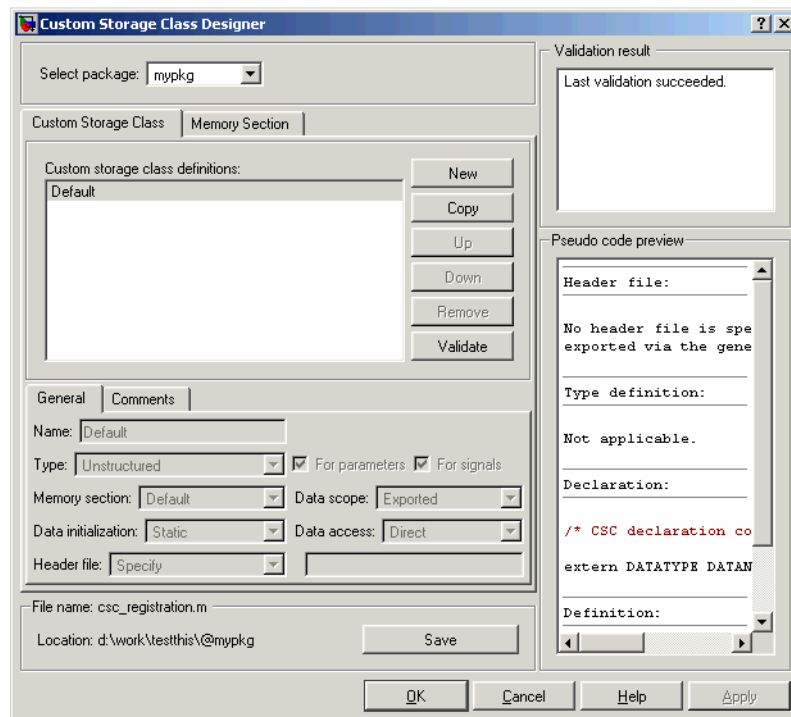
Note To create a CSC registration file for a package, the **Create your own custom storage classes for this class** option must be selected for at least one of the classes in the package.

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. 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” on page 4-15. 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.

Defining Advanced Custom Storage Class Types

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 “`GetSet` Custom Storage Class for Data Store Memory” on page 4-38) is an example of an advanced CSC that is provided with Real-Time Workshop Embedded Coder.

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 4-30.

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 CSCs” on page 4-6 for further information on instance-specific properties.

Write TLC Code for Your CSC

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

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

```
@PackageName/tlc
```

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

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

Register Custom Storage Class Definitions

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

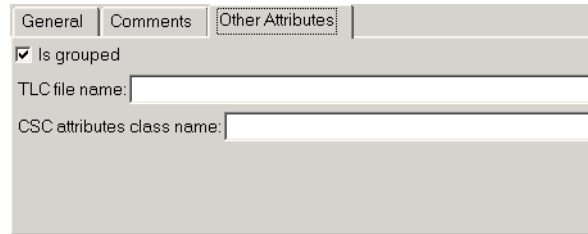
The advanced mode supports selection of an additional storage class **Type**, designated `Other`. The `Other` type is designed to support special CSC types that cannot be accommodated by the standard `Unstructured` and `FlatStructure` custom storage class types. The `Other` type cannot be assigned to a CSC except when the Custom Storage Class Designer is in advanced mode.

To register your class definitions:

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

```
cscdesigner -advanced
```

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



The screenshot shows the 'Other Attributes' pane of the Custom Storage Class Designer. It has three tabs: 'General', 'Comments', and 'Other Attributes', with 'Other Attributes' selected. The pane contains a checked checkbox labeled 'Is grouped'. Below it are two text input fields: 'TLC file name:' and 'CSC attributes class name:'. The rest of the pane is a large, empty gray area.

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

- 4 Set the properties shown on the **Other Attributes** pane. The properties are:
 - **Is grouped:** Select this option if you intend to combine multiple data objects of this CSC into a single variable in the generated code. (for example, a struct).
 - **TLC file name:** Enter the name of the TLC file corresponding to this custom storage class. The location of the file is assumed to be in the `/t1c` subdirectory for the package, so you should not enter the path to the file.
 - **CSC attributes class name:** (optional) If you created a custom attributes class corresponding to this custom storage class, enter the full name of the custom attributes class. (see “Create a Custom Attributes Class for Your CSC (Optional)” on page 4-34).

- 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

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. The instance-specific properties of the GetSet storage class are summarized in GetSet Storage Class Properties on page 4-38.

GetSet Storage Class Properties

Property	Description
GetFunction	String that specifies the name of a function call to read data.
SetFunction	String that specifies the name of a function call to write data.
HeaderFile (optional)	String that specifies the name of a header (.h) file to add as an #include in the generated code. <hr/> <p>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. Otherwise, the generated code might not compile or might function improperly.</p> <hr/>

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
```

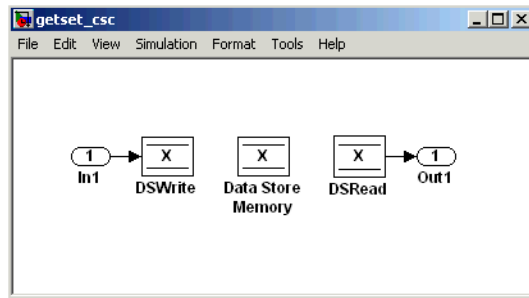
For more details about the definition of the GetSet storage class, look at its associated TLC code in the file

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

Code Generation Example

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.

Requirements and Restrictions for Use of CSCs

This section describes how to set code generation options that affect the operation of CSCs, and discusses a few restrictions on the use of CSCs.

Setting Related Code Generation Options

- During code generation, custom storage classes assigned to parameters are ignored unless the **Inline parameters** option in the **Optimization** options tab 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 active configuration set is deselected. When this option is on, data objects with custom storage classes are treated as if their storage class attribute is set to Auto.

Restrictions

The Fcn block does not support parameters with custom storage class in code generation.

Use of CSCs with Model Referencing

This section describes restrictions that apply to the use of CSC in models that use the Model Referencing feature.

In the discussion below, 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*.

In the current release, the following restrictions apply to models that use the Model Referencing feature:

- 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 releases prior to Real-Time Workshop Embedded Coder 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 4-50).

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 4-47.
- `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 4-47.

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 4-45, 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 4-45, 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 the code generated by the Real-Time Workshop	Y	Y	any	Y	Y

The final three classes, shown in BitField, Define, and Struct Storage Classes (Prior to Release 14) on page 4-46, 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 4-48 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 #include directive.	N
ImportFromFile	IncludeDelimiter	Enumerated. Defines delimiter used for filename in the #include directive. Delimiter is either double quotes (for example, #include "vars.h") or angle brackets (for example, #include <vars.h>). The default is quotes.	Y
BitField	BitFieldName	String. Defines name of bit field in which data is embedded; if unassigned, the name defaults to <i>rt_BitField</i> .	Y
Struct	StructName	String. Defines name of the struct in which data is embedded; if unassigned, the name defaults to <i>rt_Struct</i> .	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 Structures” 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 Real-Time Workshop Embedded Coder.

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 **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 Real-Time Workshop Embedded Coder, you can control the **Ignore custom storage classes** option with the check box in the **Real-Time Workshop** pane of the Configuration Parameters dialog.

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. 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 4-30 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 4-51.

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 Simulink 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:

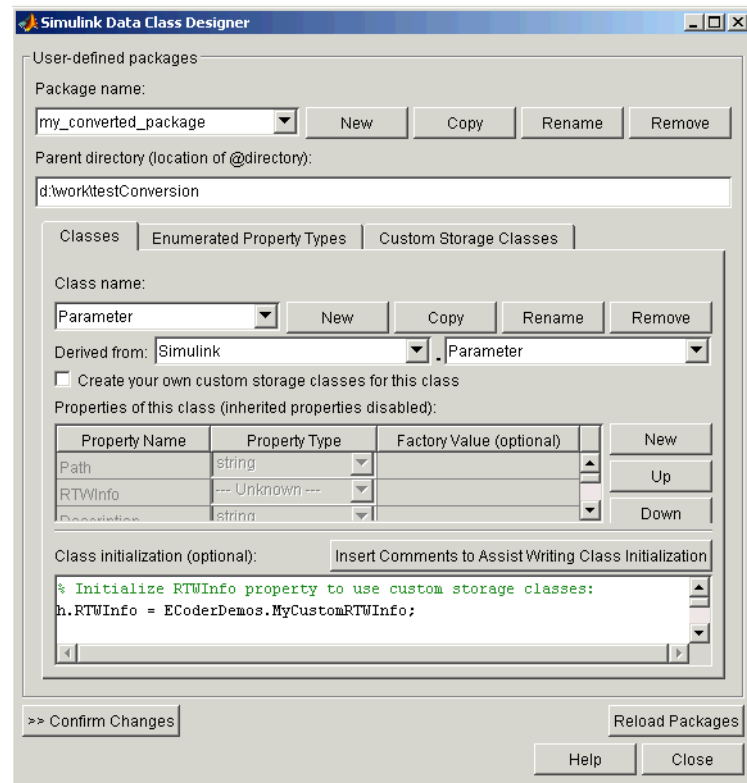
- 1 You should make a complete backup copy of the package directory before converting the package. After backing up the directory, remove the `@` prefix from the backup directory name and make sure that the backup directory is not on the MATLAB path.

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

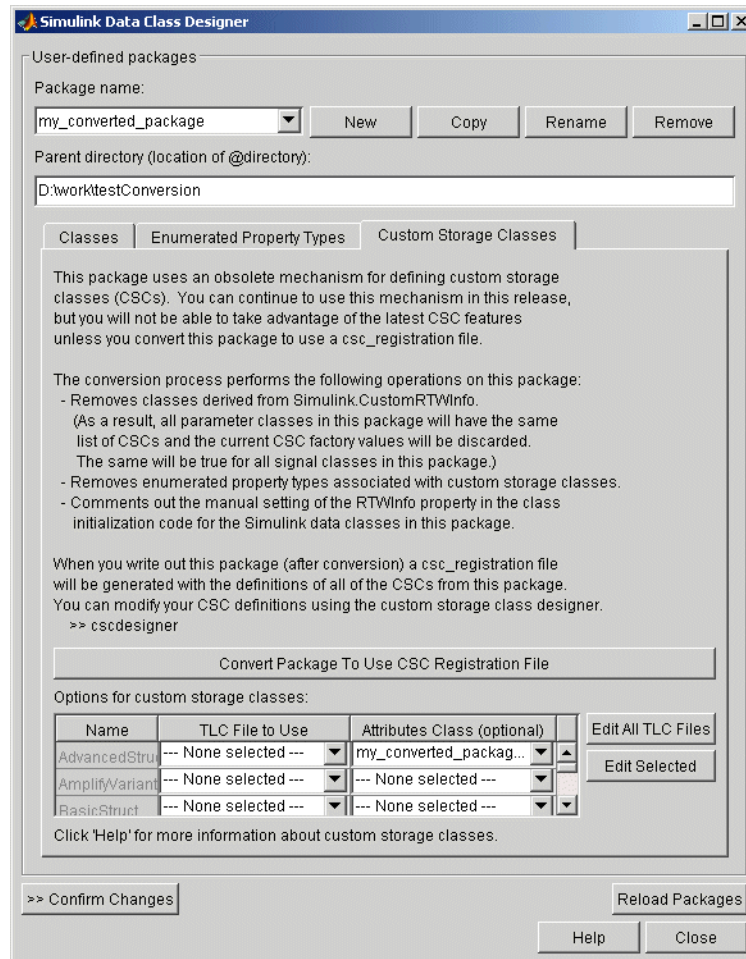
```
sldataclassdesigner
```

- 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**.
- 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`.



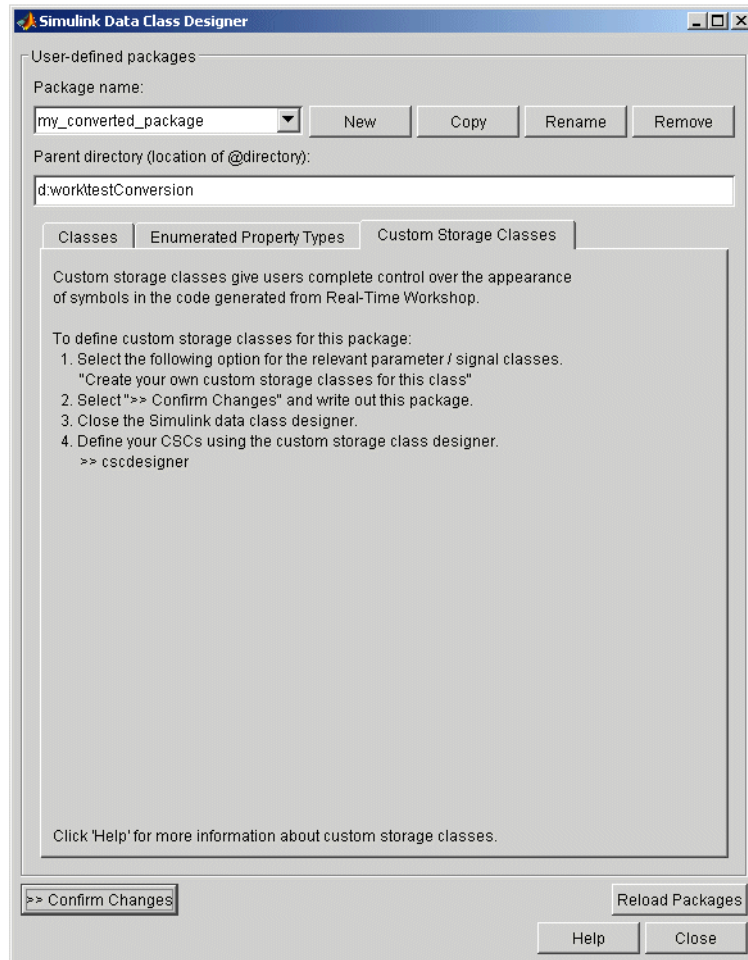
- 5 Click on the **Custom Storage Classes** pane. The pane displays a message indicating that the package contains obsolete CSC definitions, as shown in this figure.



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:

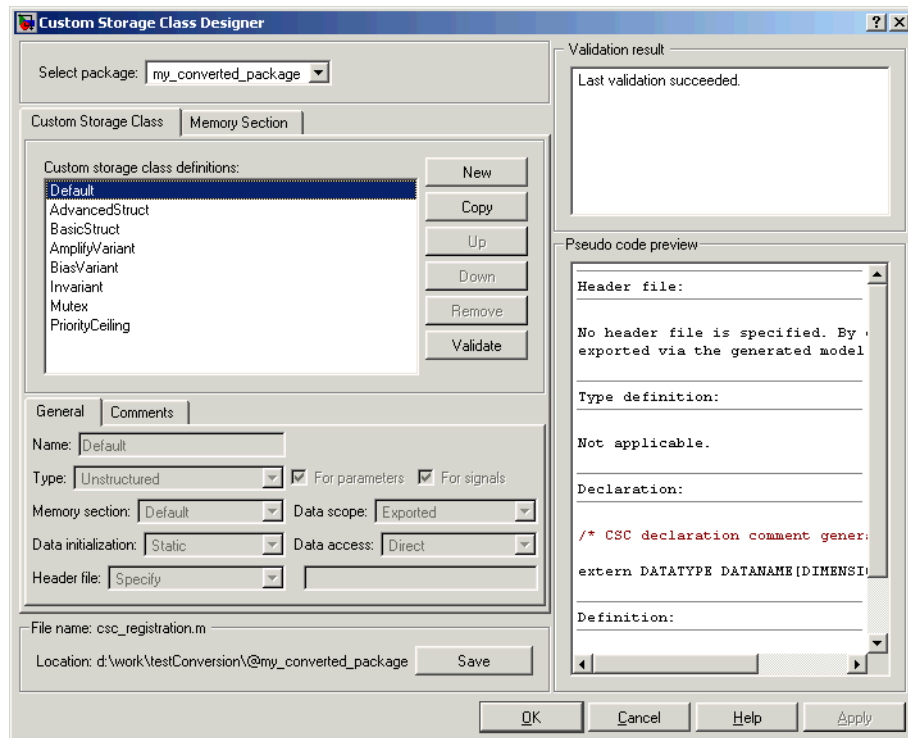
```
cscdesigner
```

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” on page 4-15 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 4-34 for more information.

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

Advanced Code Generation Techniques

Introduction (p. 5-3)

Overview of this chapter.

Code Generation with User-Defined Data Types (p. 5-5)

How to map your own data type definitions to Simulink built-in data types.

Customizing the Target Build Process with the `STF_make_rtw` Hook File (p. 5-8)

Explains the build process hook mechanism and how to use an `STF_make_rtw_hook.m` hook file to modify the build process.

Auto-Configuring Models for Code Generation (p. 5-14)

How to use the `STF_make_rtw_hook.m` hook file and supporting utilities to automate the configuration of a model during the code generation process.

Generating Efficient Code with Optimized ERT Targets (p. 5-18)

Describes auto-configuring versions of the ERT target that are optimized for fixed-point or floating-point code generation.

Custom File Processing (p. 5-26)

Customizing generated code with template files and the high-level code template API.

Optimizing Your Model with Configuration Wizard Blocks and Scripts (p. 5-53)

How to use Configuration Wizard blocks and scripts to configure and optimize code generation options quickly and easily.

Replacement of *STF_rtw_info_hook*
Mechanism (p. 5-65)

Use of the *STF_make_rtw_hook* hook file mechanism for specifying target-specific characteristics for code generation has been supplanted by the Hardware Implementation pane of the Configuration Parameters dialog. Read this section if you have created an *STF_make_rtw_hook* file for use with a custom target, prior to MATLAB Release 14.

Optimizing Task Scheduling for
RTOS Targets (p. 5-66)

Use the `rmStepTask` macro to optimize task scheduling for RTOS targets.

Introduction

This chapter describes advanced code generation features and techniques supported by Real-Time Workshop Embedded Coder. 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 5-8 describes the general mechanism for adding target-specific customizations to the build process.
 - “Auto-Configuring Models for Code Generation” on page 5-14 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 5-18.
 - “Optimizing Your Model with Configuration Wizard Blocks and Scripts” on page 5-53 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 5-26 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 5-47 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 5-53 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.

Code Generation with User-Defined Data Types

Real-Time Workshop Embedded Coder 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.

Note The names of data types defined using `Simulink.AliasType` objects are preserved in the generated code only for installations licensed for Real-Time Workshop Embedded Coder.

In general, code generated from user-defined data objects conforms to the properties and attributes of the objects as defined for use in simulation. However, the `HeaderFile` property (string) is used in code generation only. Use of this property is optional. If the `HeaderFile` property of the object is left empty, the generated code includes a `typedef` for the user-defined type in `model_types.h`. For example, for a `Simulink.NumericType` object `myfloat` with a `Category` of `double`, the generated `typedef` is

```
typedef real_T myfloat;
```

If the `HeaderFile` property is specified, a `#include` directive for the specified file is generated in `model_types.h`. 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"
```

When `HeaderFile` is specified, the header file must be located in the build process include path, and must include the appropriate `typedef` statements.

In general, 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. However, there is an exception to this rule: 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.

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

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.

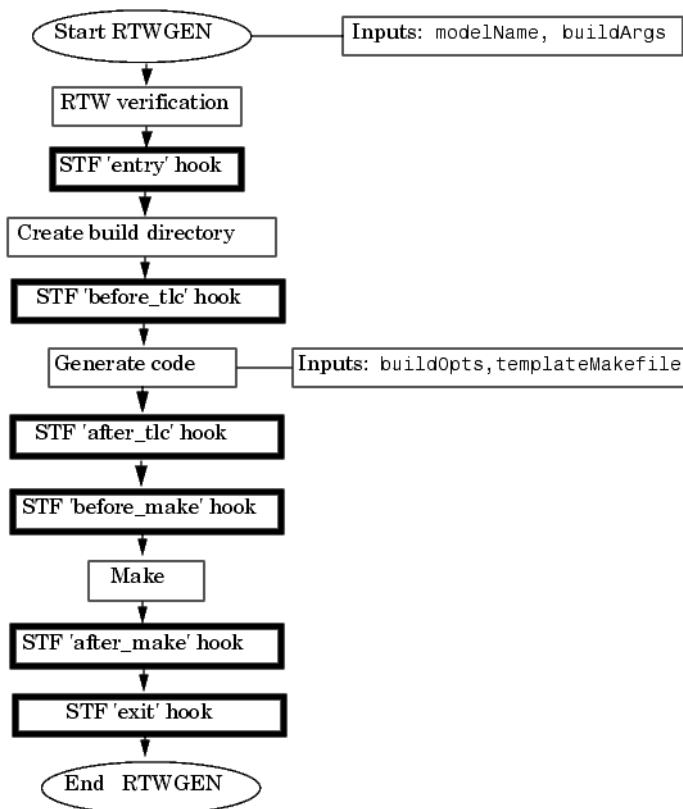
`STF_make_rtw_hook.m` Function Prototype and Arguments

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:

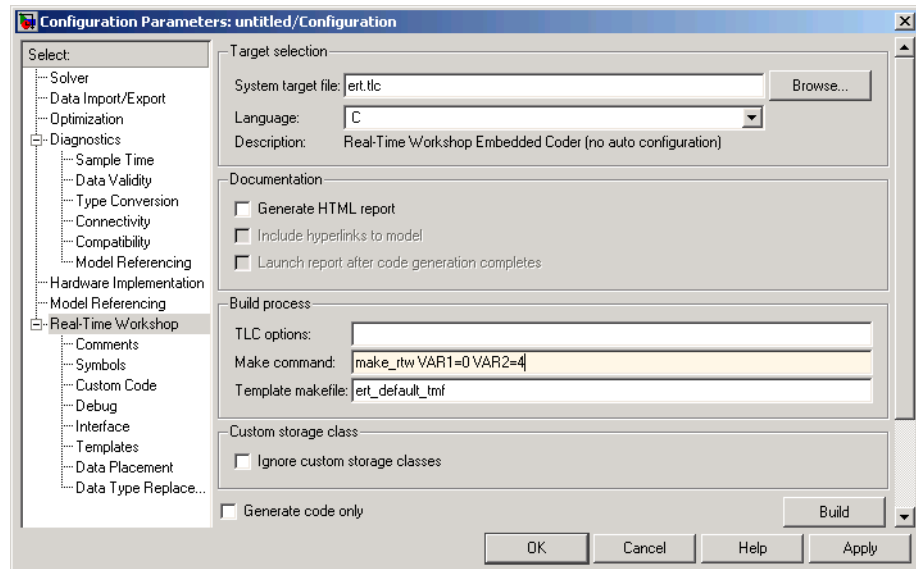
- `hookMethod`: String specifying the stage of build process from which the `STF_make_rtw_hook` function is called. The flowchart below summarizes the build process, highlighting the hook points. Valid values for `hookMethod` are `'entry'`, `'before_tlc'`, `'after_tlc'`, `'before_make'`, `'after_make'`,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



Build Process Flowchart (Hook Points Highlighted)

- rtwRoot: Reserved.
- modelName: String specifying the name of the model. Valid at all stages of the build process.
- 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 generated C or C++ files, such as *model.c* or *.cpp*, *model_data.c* or *.cpp*, and so on

- 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.



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', and 'exit', 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.

Note that the build process temporarily changes the MATLAB working directory to the build directory for stages 'before_make', 'after_make', and 'exit'. 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, 'MakeRTWSettingsObject');  
buildDirPath = getfield(makertwObj, 'BuildDirectory');
```

Using STF_make_rtw_hook.m for Your Build Procedure

To create a custom *STF_make_rtw_hook* hook file for your build procedure, copy and edit the example *ert_make_rtw_hook.m* file (located in the *matlabroot\toolbox\rtw\targets\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 5-8. 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 5-14 for an example.

Auto-Configuring Models for Code Generation

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 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

Simulink 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 Simulink, Stateflow, Real-Time Workshop, and Real-Time Workshop Embedded Coder.

Using `set_param`

The `set_param` utility can be used to assign values to model parameters, to backup and restore model settings, and to display information about model options.

To assign an individual model parameter value, 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. For a list of parameters that you can specify and their Configuration Parameters mapping, see 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 5-9). 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
```

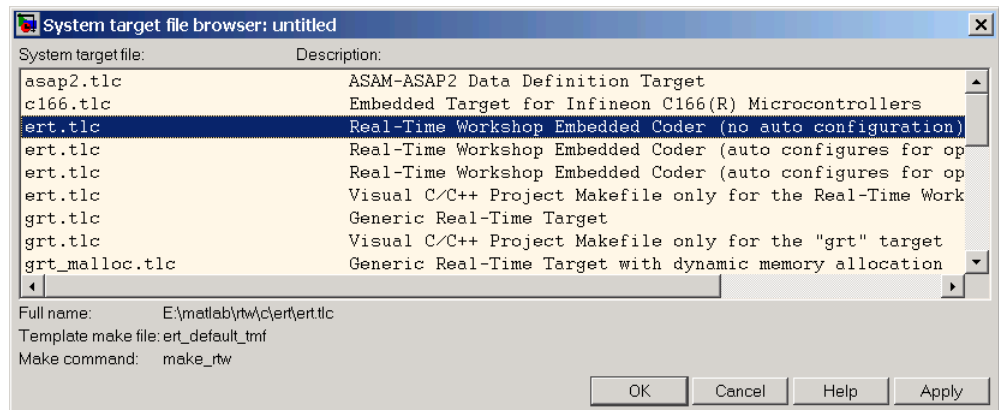
Using the Auto-Configuration Utilities

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 5-8).
- 2** Reconfigure the `set_param` calls within `ert_config_opt.m` to suit your application needs.

Generating Efficient Code with Optimized ERT Targets

To make it easier for you to generate code that is optimized for your target hardware, Real-Time Workshop Embedded Coder 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. (See the “Hardware Implementation Pane” section of the Simulink 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

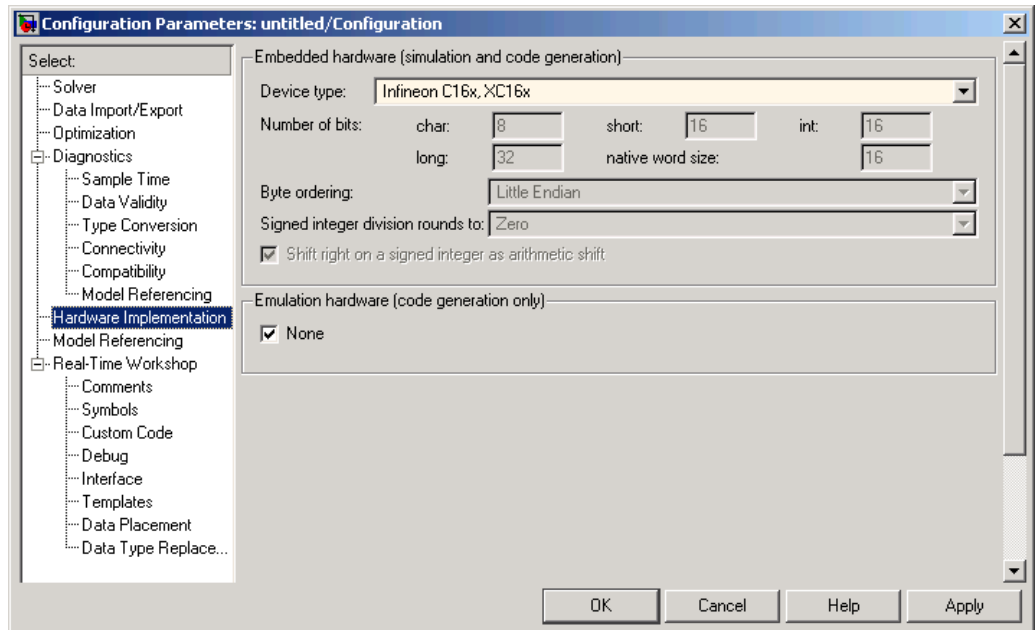
```
Real-Time Workshop Embedded Coder (auto configures for optimized fixed-point code)
```

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

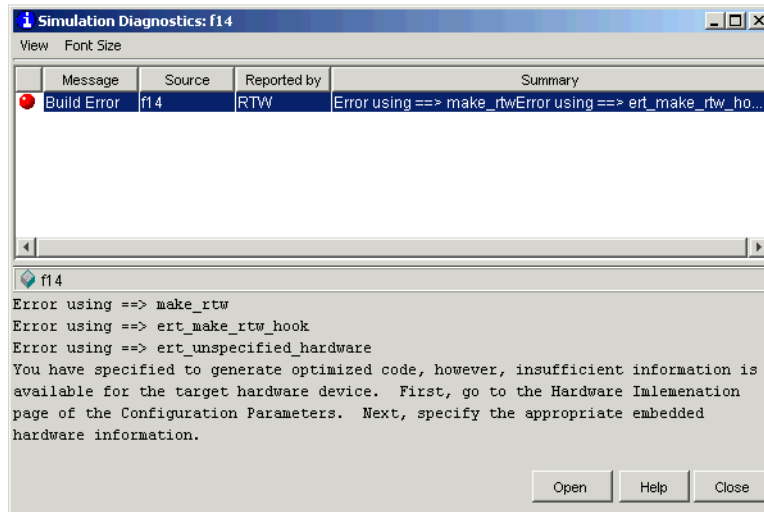
The optimized fixed-point ERT target passes in the command `optimized_fixed_point=1` to the target selection process, and also to the build process with the **Make command** field of the **Real-Time Workshop** pane of the Configuration Parameters dialog. This in turn invokes the M-file `ert_config_opt.m`, 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 5-14 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 type** (or define a Custom device type) in the upper panel of the **Hardware Implementation** pane of the Configuration Parameters dialog, and set the other properties appropriately for your target. In the figure below, the **Device type** field is configured for the Infineon C16x, XC16x and XC16x microprocessor family.



If the **Device type** field is set to Unspecified, 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

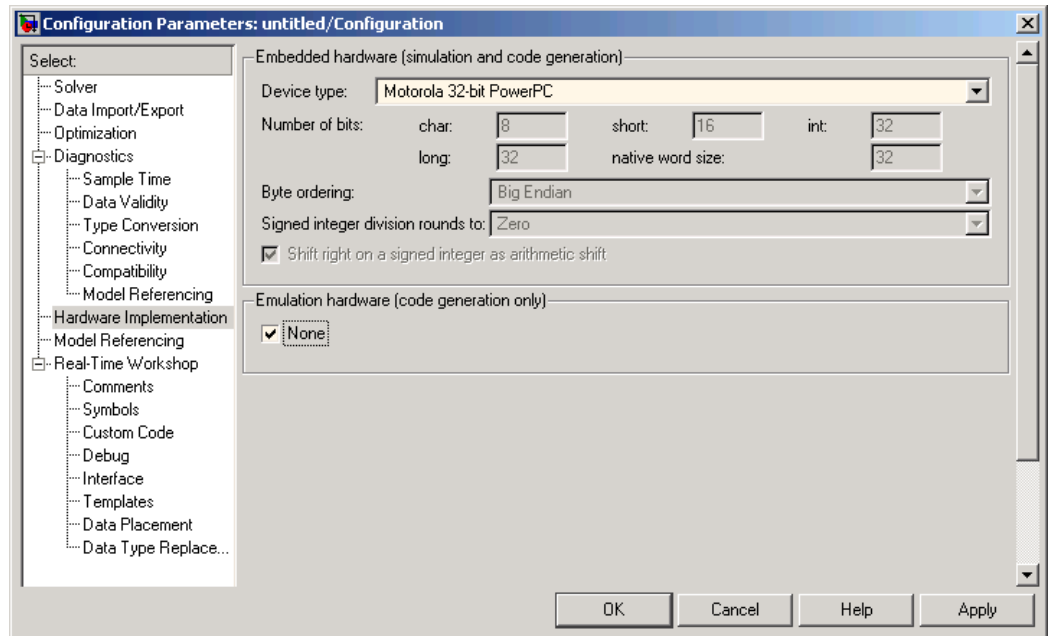
Real-Time Workshop Embedded Coder (auto configures for optimized floating-point code)

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

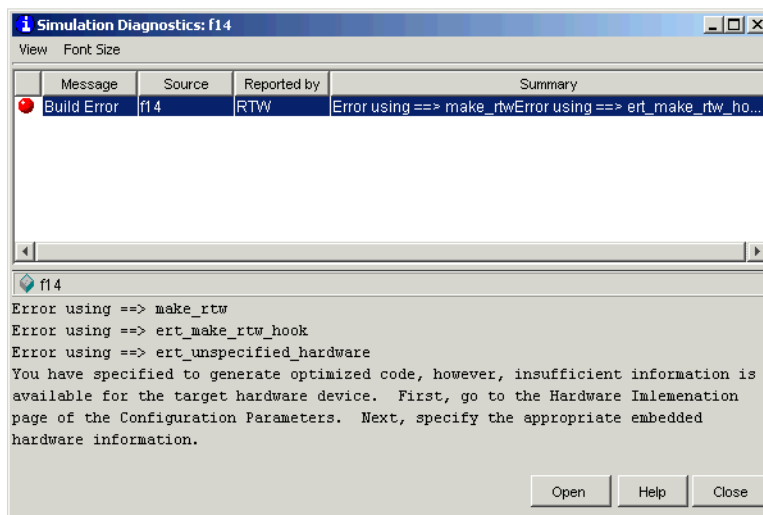
The optimized floating-point ERT target passes in the command `optimized_floating_point=1` to the target selection process, and also to the build process with the **Make command** field of the **Real-Time Workshop** pane of the Configuration Parameters dialog. This in turn invokes the M-file `ert_config_opt.m`, 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 5-14 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 type** (or define a Custom device type) in the upper panel of the **Hardware Implementation** pane of the Configuration Parameters dialog, and set the other properties appropriately for your target. In the figure below, the **Device type** field is configured for the Motorola PowerPC family of microprocessors.



If the **Device type** field is set to Unspecified, 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

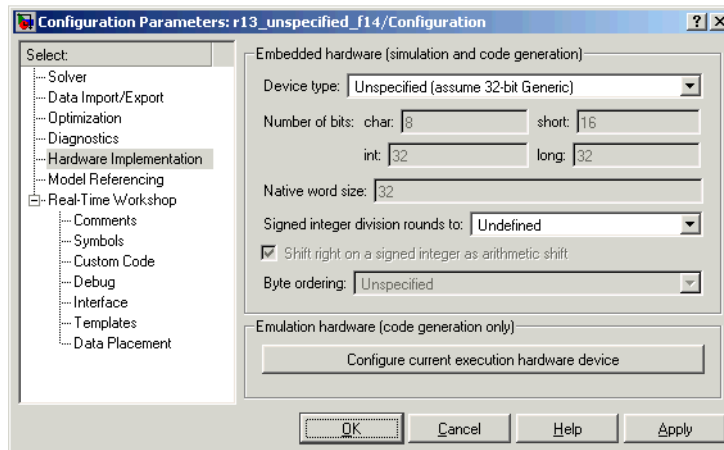
This section describes how to use the optimized ERT targets in code generation.

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 type** (or define a Custom device type) in the upper panel of the **Hardware Implementation** pane of the Configuration Parameters dialog, and set the other properties appropriately for your target (see “Optimized Fixed-Point ERT Target” on page 5-19 and “Optimized Floating-Point ERT Target” on page 5-21). 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 type** defaults to **Unspecified**, 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.

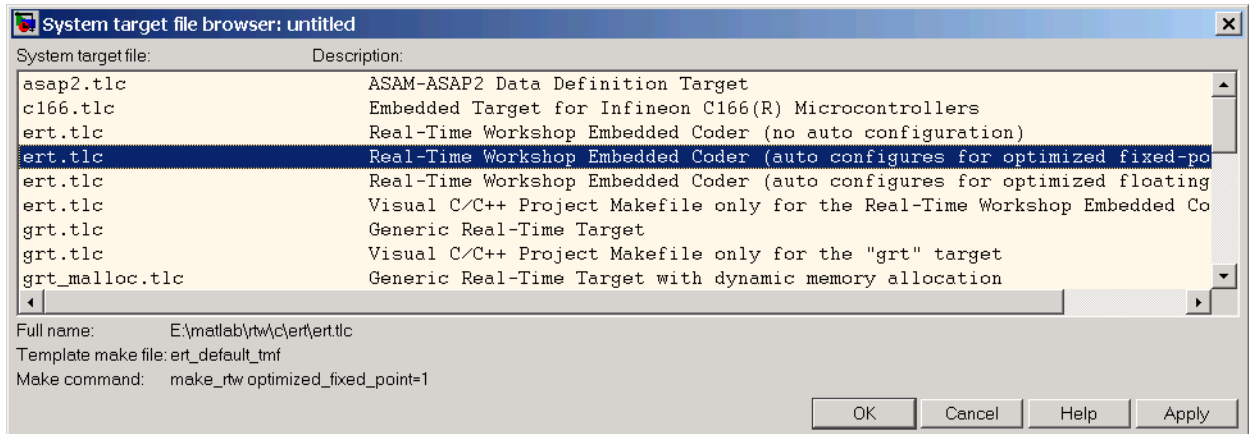


In this case you should click the button to set the **Emulation hardware** properties to a valid (default) state, and save the model.

Generating Code

To generate code using one of the optimized ERT targets:

- 1 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 Save the model.
- 3 Initiate the build process.
- 4 If your model's **Hardware Implementation** parameters are not configured correctly, an error message is displayed. See “Configuring Hardware Implementation Properties” on page 5-23 to learn how to correct this.
- 5 During code generation, the auto-configuration code executes. This is reported in a message similar to the following:

```
*** Auto configuring 'optimized_fixed_point' for model 'ecdemo' as specified by:
D:\test_install\r13spi_bash\toolbox\rtw\targets\ecoder\ert_config_opt.m
*** Overwriting model settings if they do not yield optimized code.
```

- 6 Other than the above message, the build process executes normally, reporting the usual progress and completion messages.

Custom File Processing

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 5-29.
- 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 5-44.

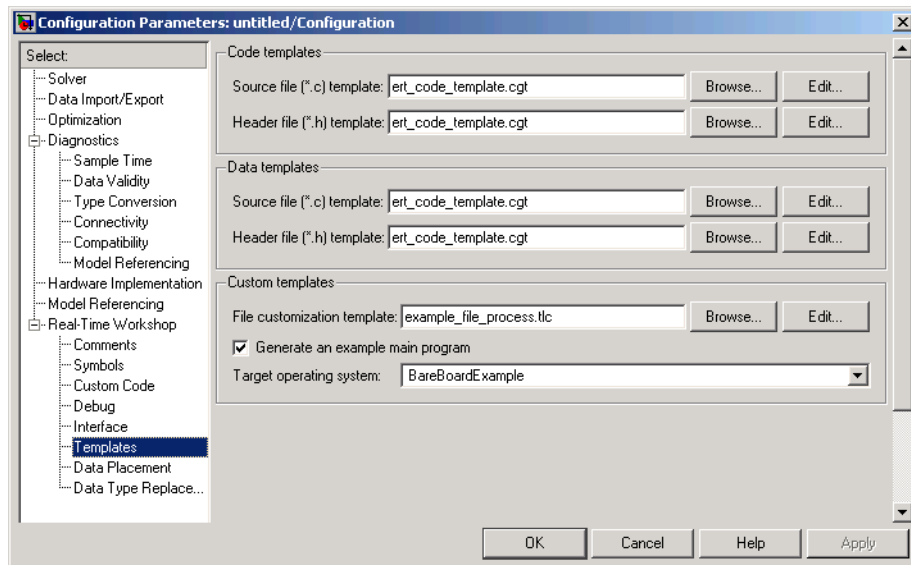
- *Custom file processing (CFP) templates*: A CFP template is a TLC file that manages the process of custom code generation. The primary purpose of a CFP template is to assemble code to be generated into buffers, and to call the code template API to emit the buffered code into specified sections of generated 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 5-33.

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 Real-Time Workshop Embedded Coder. 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 5-28 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. By default, the code template for source files is `matlabroot\toolbox\rtw\targets\ecoder\ert_code_template.cgt`.
- 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 code template for source 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 CFP template or CGT file. **Edit** opens the specified CFP template into the MATLAB editor, where you can customize it.

- 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 `example_file_process.tlc`.

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 5-47.
- 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

Real-Time Workshop Embedded Coder 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 5-47 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>
```

Real-Time Workshop Embedded Coder 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 5-30). You can also define *custom* tokens and add them to the code insertion section (see “Generating a Custom Section” on page 5-42).

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 5-33.

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 5-32.

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 5-47 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 5-31 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
Enums	Enumerated types section
Definitions	Place data definitions here (for example, <code>double x = 3.0;</code>)

Built-In CGT Tokens and Corresponding Code Sections (Continued)

Token / Section Name	Description
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 5-32.

It is important to note that the sections and subsections listed in Subsections Defined for Built-In Sections on page 5-32 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 5-32. See “Generating a Custom Section” on page 5-42 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.
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 IntrinsicTypes 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.

Subsections Defined for Built-In Sections (Continued)

Section	Subsections	Subsection Description
Types	Typedefs	typedef section. Typedefs can depend on any previously defined type
Enums	N/A	
Definitions	N/A	
Declarations	N/A	
Functions		C or C++ functions
Functions	CompilerErrors	#warning directives
Functions	CompilerWarnings	#error 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 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 CFP Template” on page 5-35.
- TLC files supporting generation of single-rate and multi-rate main program modules (see “Customizing Main Program Module Generation” on page 5-40).

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 5-27).

CFP Template Structure

A CFP template imposes a simple structure on the code generation process. The template, in conjunction with a 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 5-31 and Subsections Defined for Built-In Sections on page 5-32.

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 5-32.

Determine the `section` argument as follows:

- If Subsections Defined for Built-In Sections on page 5-32 defines no subsections for a given section, use the section name as the `section` argument.
 - If Subsections Defined for Built-In Sections on page 5-32 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 5-42).
- `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 CFP Template” on page 5-35, provides typical usage examples.

Generating Source and Header Files with a 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, `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
```

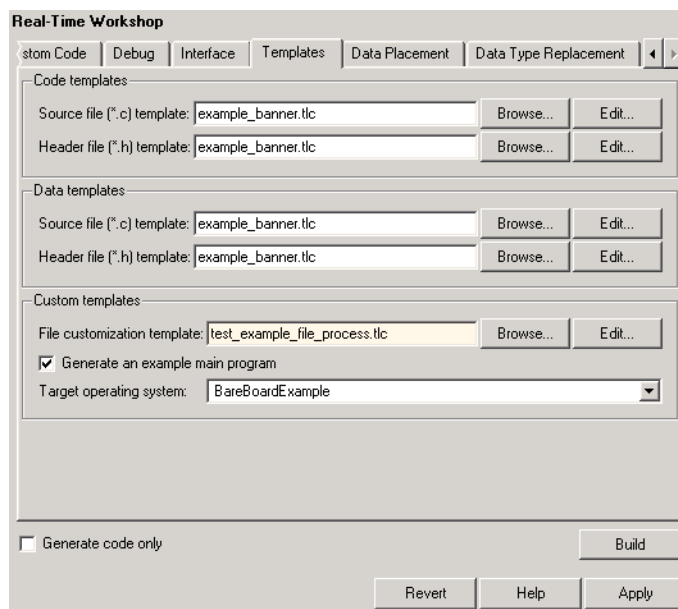
It should now read:

```
%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 ecdemo 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 ecdemo 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` or `.cpp`, `timestwo.h`, and `ert_main.c` or `.cpp`. 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 `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 hFile = LibCreateSourceFile("Header", "Custom", "timestwo")
...
%assign cFile = LibCreateSourceFile("Source", "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 5-32.

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 */
real_T timestwofcn(real_T 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 `ecdemo.h` file to see the generated `#define` directive.

Customizing Main Program Module Generation. Normally, the ERT target follows the **Generate an example main program** and **Target operating system** options to determine how to generate an `ert_main.c` or `.cpp` module (if any). 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 multi-rate main program module for a bareboard target environment. Code is generated by a single TLC function, `FcnMultiTaskingMain`.

In the example CFP template, the following code generates either a single- or multi-tasking `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 2-12 and “Guidelines for Modifying `rt_OneStep`” on page 2-18 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 excerpts illustrate the addition of a custom token and section associated with the built-in `Includes` section.

First, the token `Myincludes` is added to the code insertion section of the CGT file.

```
%<Includes>
%<Myincludes>
%<Defines>
%<Types>
%<Enums>
```

```
%<Definitions>
%<Declarations>
%<Functions>
```

In the CFP file, two include directives are generated into a buffer.

```
%openfile MyTmp
#include "moretables1.h"
#include "moretables2.h"
%closefile MyTmp
```

The following function call declares an association between the built-in section `Includes` and the custom section `Myincludes`. In effect, `Myincludes` is a subsection of `Includes`.

```
%<LibAddSourceFileCustomSection(modelC,"Includes","Myincludes")>
```

The following call to `LibSetSourceFileCustomSection` directs the code in the `MyTmp` buffer to the desired section of the generated file. `LibSetSourceFileCustomSection` is syntactically identical to `LibSetSourceFileSection`.

```
%<LibSetSourceFileCustomSection(modelC,"Myincludes",MyTmp) >
```

In the generated code, the include directives generated to the custom section appear after other code directed to `Includes`.

```
#include "ecdemo.h"
#include "ecdemo_private.h"
#include "moretables1.h"
#include "moretables2.h"
```

Note that the placement of the custom token in this example is arbitrary. By locating `%<Myincludes>` after `%<Includes>`, the CGT file ensures only that the `Myincludes` code appear after `Includes` code.

Code Template API Summary

Code Template API Functions on page 5-44 summarizes the code template API. See the source code in `codetemplatelib.tlc` for detailed information on the arguments, return values, and operation of these calls.

Code Template API Functions

Function	Description
<code>LibGetNumSourceFiles</code>	Returns the number of created source files (.c or .cpp and .h).
<code>LibGetSourceFileTag</code>	Returns <code><filename>_h</code> and <code><filename>_c</code> for header and source files, respectively, where <code>filename</code> is the name of the model file.
<code>LibCreateSourceFile</code>	Creates a new C or C++ file and returns its reference. If the file already exists, simply returns its reference.
<code>LibGetSourceFileFromIdx</code>	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.
<code>LibSetSourceFileSection</code>	Adds to the contents of a specified section within a specified file (see also “CFP Template Structure” on page 5-34).
<code>LibGetSourceFileSection</code>	Retrieves the contents of a file section. See the code for <code>LibSetSourceFileSection</code> for list of valid sections.
<code>LibIndentSourceFile</code>	Indents a file with the <code>c_indent</code> utility of Real-Time Workshop (from within the TLC environment).
<code>LibCallModelInitialize</code>	Returns code for calling the model’s <code>model_initialize</code> function (valid for ERT only).
<code>LibCallModelStep</code>	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

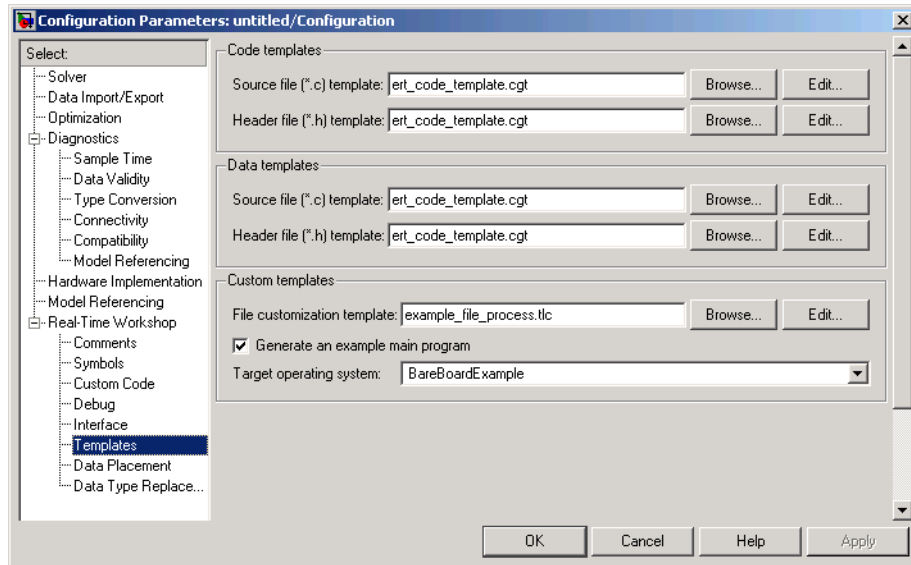
Using CGT files, you can specify custom *file banners* to be inserted into generated code files. File banners are comment sections in the header and trailer portions of a generated file. You can use these banners to add a company copyright statement, specify a special version symbol for your configuration 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 5-48).



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 code 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 5-48).
- 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 not inside the MATLAB directory structure. Note that this directory must be 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 edit the associated **Source file (.c) template** or **Header file (.h) template** field to match the new filename.
- 10 Edit and customize the CGT file as needed (See “Customizing a CGT File for Custom Banner Generation” on page 5-50). Before exiting the MATLAB editor, save your changes to the CGT file.
- 11 Click **Apply** to update the configuration set.
- 12 Save your model.

- 13 Generate code. Examine the generated source and/or header files to confirm that they contain the banners specified by the template(s).

Customizing a CGT File for Custom Banner Generation

This section describes the sections of a CGT file you need to modify for custom file banner generation. For a more detailed description of CGT files, see “Code Generation Template (CGT) Files” on page 5-29.

Custom file banner generation requires modification of one or more of the following CGT file sections:

- **Header section:** This section contains comments and tokens for use in generating a 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>
 *
 * You can customize this banner by specifying a different template.
 */
```

- **Trailer section:** This section contains comments and tokens for use in generating a 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.
 *
 * You can customize this file trailer by specifying a different template.
 *
 * [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 5-52 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 2003 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 an actual file banner generated from the ecdemo model using the above template.

```

/*
 * File: ecdemo.c
 * -----
 * Copyright 2003 ABC Corporation, Inc.
 * -----
 * Real-Time Workshop code generated for Simulink model ecdemo.
 *
 * Model version                : 1.188
 * Real-Time Workshop file version : 6.0 (R14 Prerelease) 13-Nov-2003
 * Real-Time Workshop file generated on : Tue Nov 18 16:46:48 2003
 * TLC version                  : 6.0 (Nov 15 2003)
 * C source code generated on    : Tue Nov 18 16:46:52 2003
 *
 *
 */

```

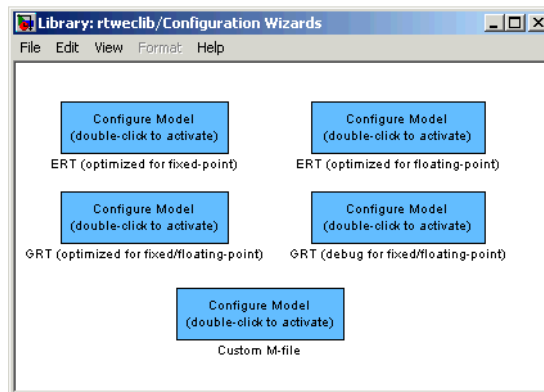
Summary of Tokens for File Banner Generation

FileName	Name of the generated file (for example, "ecdemo.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

Real-Time Workshop Embedded Coder 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 four preset Configuration Wizard blocks, and a Configuration Wizard block you can customize. These are shown in the figure below.



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 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.
- 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 5-14 and “Generating Efficient Code with Optimized ERT Targets” on page 5-18) 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.

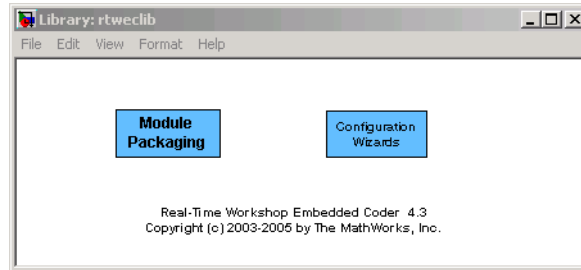
Adding a Configuration Wizard Block to Your Model

This section describes how to add one of the preset Configuration Wizard blocks to a model.

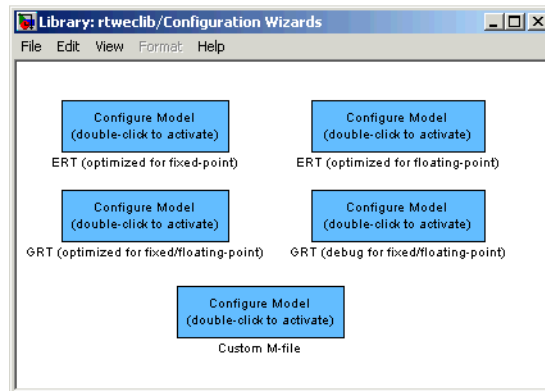
The Configuration Wizard blocks are available in the Real-Time Workshop Embedded Coder block library. To use a Configuration Wizard block:

- 1 Open the model that you want to configure.
- 2 Open the Real-Time Workshop Embedded Coder library by typing the command `rtweclib`.

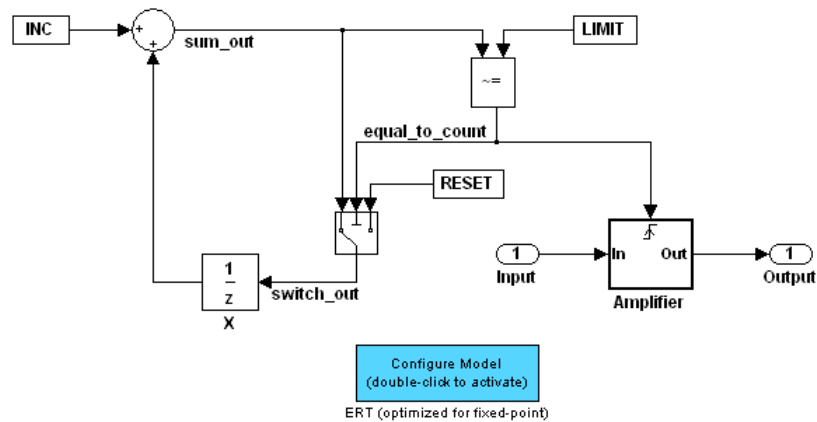
3 The top level of the library is shown below.



4 Double-click the Configuration Wizards icon. The Configuration Wizards sublibrary opens, as shown below.

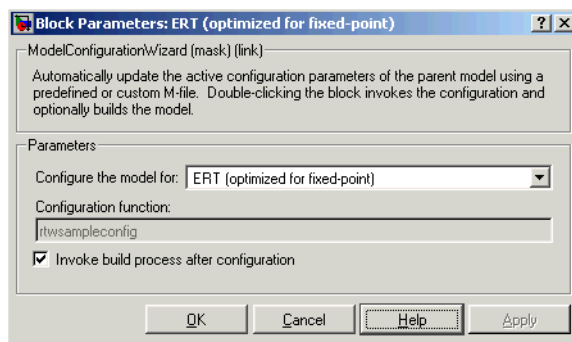


- Select the Configuration Wizard block you want to use and drag and drop it into your model. In the figure below, the ERT (optimized for fixed-point) Configuration Wizard block has been added to the model.



- 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.

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 5-57.

- 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 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 that 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:

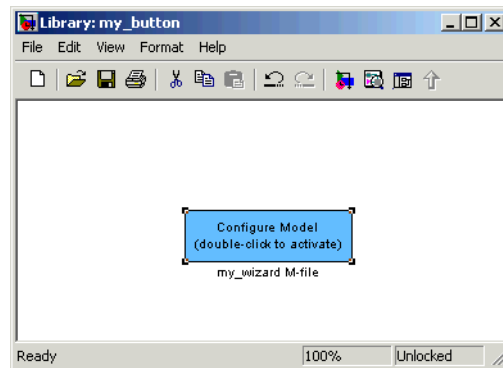
```
disp('Custom Configuration Wizard Script completed.');
```

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:

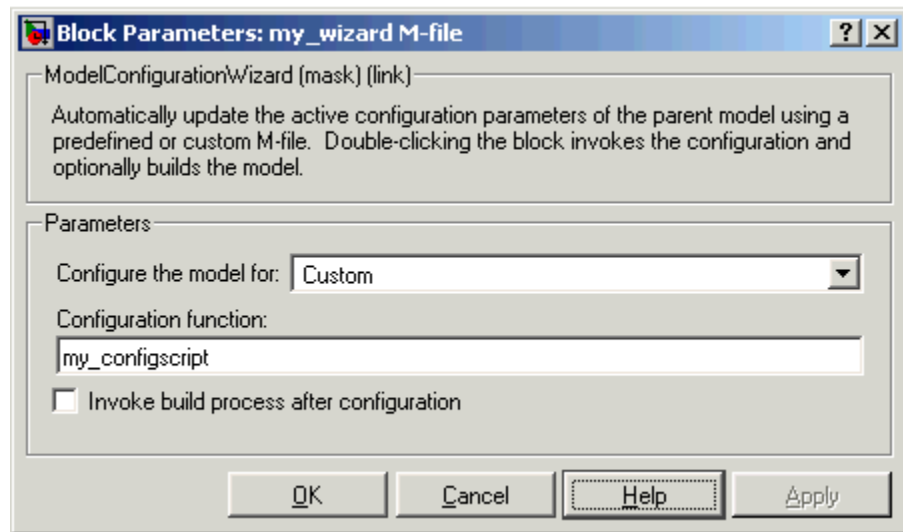
- 1 Open the Real-Time Workshop Embedded Coder library and the Configuration Wizards sublibrary, as described in “Adding a Configuration Wizard Block to Your Model” on page 5-54.
- 2 Select **New Library** from the **File** menu of the Configuration Wizards sublibrary window. An empty library window 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.
- 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 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.
- 3 Observe that the vdp demo is configured, by default, for the GRT target. Close the Configuration Parameters dialog.
- 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 and view the **Real-Time Workshop** options 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. Simulink 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). 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 **Generate HTML report** option:

```
if strcmp(get_param(cs, 'GenerateReport'), 'on')
    ...
end
```


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 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. Using this dialog, 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 “Hardware Implementation Options” section of the Real-Time Workshop documentation.

Optimizing Task Scheduling for RTOS Targets

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 multi-rate, 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 bare-board 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 multi-rate, multitasking model, the event flags are maintained by the `model_SetEventsForThisBaseStep` function. `model_SetEventsForThisBaseStep` invokes the macro `rtmStepTask` to test the value of each counter. `rtmStepTask` returns `TRUE` when a counter indicates that a task's sample period has elapsed. When this occurs, `model_SetEventsForThisBaseStep` sets the event flag for that task.

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 bare-board 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. An RTOS target can eliminate the call to `model_SetEventsForThisBaseStep`, and examine the counters by invoking `rtmStepTask` directly.

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`. The following example, from `ertmainlib.tlc`, is designed for the VxWorks RTOS. A loop iterates over each subrate task. `rtmStepTask` is called for each task. If `rtmStepTask` returns `TRUE`, the VxWorks `semGive()` function is called, and VxWorks 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 the 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
```

Requirements, Restrictions, Target Files

Requirements and Restrictions
(p. 6-2)

Conditions your model must meet
for use with Real-Time Workshop
Embedded Coder.

System Target File and Template
Makefiles (p. 6-4)

Summary of control files used by
Real-Time Workshop Embedded
Coder.

Requirements and Restrictions

- For code generation with Real-Time Workshop Embedded Coder, configure your model for the following options on the **Solver** page of the Simulation Parameters dialog:
 - **Type:** fixed-step
 - **Solver:** You can select any available solver algorithm.
 - **Mode:** You must select the SingleTasking or Auto solver mode when the model is single-rate. 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 specify the **Application lifespan (days)** parameter properly. (See “Blocks That Depend on 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
 - 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 and Fixed-Point block libraries, execute the following command at the MATLAB command line:

```
showblockdatatypetable
```

Refer to the “Recommended for Production Code?” column of the table.

- You can use both inlined and non-inlined S-functions with Real-Time Workshop Embedded Coder. 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.

System Target File and Template Makefiles

The Real-Time Workshop Embedded Coder system target file is `ert.tlc`.

Real-Time Workshop provides template makefiles for the Real-Time Workshop Embedded Coder in the following development environments:

- `ert_bc.tmf` — Borland C
- `ert_intel.tmf` — Intel compiler
- `ert_lcc.tmf` — LCC compiler
- `ert_tornado.tmf` — Tornado (VxWorks)
- `ert_unix.tmf` — UNIX host
- `ert_vc.tmf` — Visual C
- `ert_msvc.tmf` — Visual C, project file only
- `ert_watc.tmf` — Watcom C

A

- absolute time 3-28
- ASAP2 file generation 3-34
- auto-configuring targets
 - purpose of 5-14

C

- code generation options
 - Application lifespan (days) 3-43
 - Create Simulink (S-Function) block 3-33
 - Custom comments 3-18
 - Data exchange 3-33
 - Data initialization 3-42
 - External mode 3-51
 - File customization template 3-36
 - Fixed-point exception protection 3-43
 - Generate an example main program 3-36
 - Generate HTML report 3-16
 - Generate reusable code 3-30
 - Generate scalar inlined parameters 3-21
 - GRT compatible call interface 3-30
 - Ignore custom storage classes 3-17
 - Include comments 3-17
 - Include hyperlinks to model 3-16
 - Launch report after coder generation
 - completes 3-16
 - MAT-file logging 3-33
 - clearing 3-47
 - MAT-file variable name modifier 3-33
 - Maximum identifier length 3-21
 - Minimum mangle length 3-21
 - Parameter structure 3-41
 - Pass root-level I/O as 3-31
 - Requirements in block comments 3-19
 - Reusable code error diagnostic 3-31
 - Simulink block descriptions 3-18
 - Simulink data object descriptions 3-18
 - Single output/update function 3-30
 - clearing 2-30

- Stateflow object descriptions 3-19
- Support absolute time 3-28
- Support complex numbers 3-28
- Support continuous time 3-29
 - for using continuous time blocks 3-5
 - limitations 3-57
- Support floating-point numbers 3-28
- Support non-finite numbers 3-28
- Support non-inlined S-functions 3-29
- Suppress error status in real-time model
 - data structure 3-32
- Symbol format 3-20
- Target floating-point math
 - environment 3-27
- Terminate function required 3-30
- code generation report 3-53
- code modules, generated 2-4
- code templates
 - example of use 5-35
 - generating code with 5-35
 - structure of 5-34
 - summary of API 5-44
- code, user-written 2-7
- Configuration Parameters dialog box 3-4
- Configuration Wizard buttons 5-53
- custom code generation
 - of file banners 5-47
 - with code templates 5-33
- custom file processing (CFP) template 3-36
- custom storage classes
 - assigning to data 4-49
 - code generation with 4-49
 - instance-specific attributes 4-47

D

- Data exchange options
 - External mode 3-34
 - Generate ASAP2 file 3-34

- Generate C API for
 - parameters/signals 3-34
- data initialization
 - of floats and doubles 3-42
 - of internal states 3-42
 - of root-level I/O ports 3-42
- data structures
 - real-time model 2-2
- data templates 3-35
- demos for Real-Time Workshop Embedded Coder 1-8

E

- elapsed time 3-28
- entry points, model 2-22
- ERT target
 - optimized for fixed-point 5-18
 - optimized for floating-point 5-18
- ert_main.c 2-26
- ert_main.cpp 2-26
- External mode support 3-51

F

- file banners, generation of 5-47
- file packaging 2-4

G

- generated code
 - modules 2-4
- GetSet custom storage class 4-38

H

- Hardware Implementation parameters
 - configuration of 3-45
- hook files

- STF_make_rtw_hook
 - auto-configuring models with 5-15
 - customizing build process with 5-8
- HTML code generation report 3-53

I

- installation of Real-Time Workshop Embedded Coder 1-7
- integer-only code 3-47
- integer-only code generation 3-47
- interrupts, servicing 2-11

M

- main program (ert_main)
 - generated 2-8
 - modifying 2-12
 - operation of 2-12
 - static module 2-26
 - VxWorks example 2-20
- math, floating-point 3-27
- model entry points 2-22
 - model_initialize 2-24
 - model_SetEventsForThisBaseStep 2-25
 - model_step 2-22
 - model_terminate 2-24
- modifying rt_OneStep 2-18

N

- name mangling 3-23

P

- parameter structure
 - hierarchical 3-41
 - non-hierarchical 3-41
- program execution
 - main program 2-12

- rt_OneStep 2-13
 - multi-rate multitasking operation 2-15
 - multi-rate single-tasking operation 2-18
 - reentrancy 2-17
 - single-rate single-tasking operation 2-14
- pure integer code 3-47
 - and external mode 3-52

R

- rate grouping 2-16
- real-time model data structure 2-2
- reentrant code 3-30
- requirements for Real-Time Workshop
 - Embedded Coder programs 6-2
- restrictions on Real-Time Workshop Embedded Coder programs 6-2
- reusable code 3-30

S

- S-function wrapper generation 3-56

- option for 3-33
- solver modes, permitted 2-13
- source code files, generated 2-4
- stack space allocation 3-49
- STF_make_rtw_hook function
 - arguments to 5-9
- symbol format tokens 3-22
- system target files 6-4

T

- task identifier (tid) 2-16
- template makefiles 6-4
- tid 2-16
- timer interrupts 2-11

V

- virtualized output port optimization 3-48
- VxWorks deployment example 2-20