Real-Time Workshop® Embedded Coder

For Use with Real-Time Workshop

Modeling

Simulation

Implementation

User's Guide

Version 3

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 Phone

508-647-7001 Fax

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

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

Real-Time Workshop Embedded Coder User's Guide

Printing History: July 2002 Online only Version 3.0 (Release 13)

Contents

3 Code Generation Options and Optimizations

4 Custom Storage Classes

5 Requirements, Restrictions, Target Files

A Generating ASAP2 Files

Preface

This section includes the following topics:

Prerequisites

This document assumes you have basic familiarity with MATLAB®, Simulink®, and Real-Time Workshop®. Minimally, you should read and work through all tutorials in the Real-Time Workshop documentation.

Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the Real-Time Workshop Embedded Coder. They are listed in the table below.

The Real-Time Workshop Embedded Coder *requires* these products:

- MATLAB 6.5 (Release 13)
- Simulink 5.0 (Release 13)
- Real-Time Workshop 5.0 (Release 13)

For more information about any of these products, see either:

- The online documentation for that product if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the "products" section

Note The toolboxes listed below all include functions that extend the capabilities of MATLAB. The blocksets all include blocks that extend the capabilities of Simulink.

Product	Description
Communications Blockset	Design and simulate communication systems
Control System Toolbox	Design and analyze feedback control systems
Dials & Gauges Blockset	Monitor signals and control simulation parameters with graphical instruments
DSP Blockset	Design and simulate DSP systems

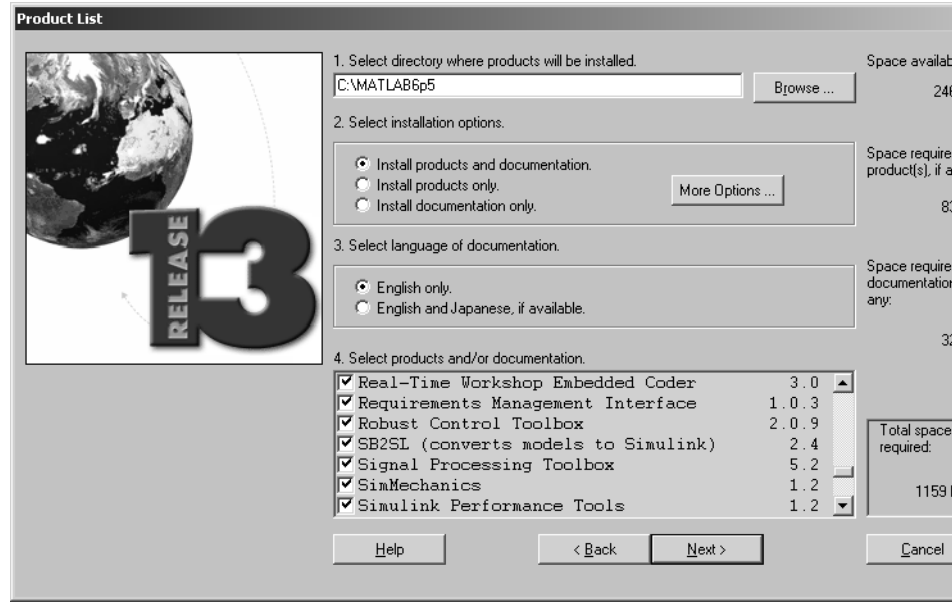
Product	Description
Embedded Target for Motorola MPC555	Generate Real-Time Workshop Embedded Coder production code for the Motorola MPC555
Embedded Target for the TI TMS320C6000™ DSP Platform	Deploy and validate DSP designs on Texas Instruments C6000 digital signal processors
Fixed-Point Blockset	Design and simulate fixed-point systems
Fuzzy Logic Toolbox	Design and simulate fuzzy logic systems
MATLAB Link for Code Composer Studio™ Development Tools	Use MATLAB with RTDX™-enabled Texas Instruments digital signal processors
Nonlinear Control Design Blockset	Optimize design parameters in nonlinear control systems
Real-Time Windows Target	Run Simulink and Stateflow models on a PC in real time
Real-Time Workshop	Generate C code from Simulink models
SimPowerSystems	Model and simulate electrical power systems
Simulink	Design and simulate continuous- and discrete-time systems
Stateflow®	Design and simulate event-driven systems
Stateflow Coder	Generate C code from Stateflow charts
Statistics Toolbox	Apply statistical algorithms and probability models
xPC Target	Perform real-time rapid prototyping using PC hardware
xPC Target Embedded Option	Deploy real-time applications on PC hardware

Installing the Real-Time Workshop Embedded Coder

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

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

As the installation process proceeds, it displays a dialog box similar to the one below, letting you indicate which products to install.



The Real-Time Workshop Embedded Coder has certain product prerequisites that must be met for proper installation and execution.

Licensed Product	Prerequisite Products	Additional Information
Simulink	MATLAB 6.5 (Release 13)	Allows installation of Simulink.
The Real-Time Workshop	Simulink 5.0 (Release 13)	Requires Borland C, LCC, Visual C/C++, or Watcom C compiler to create MATLAB MEX-files on your platform.
The Real-Time Workshop Embedded Coder	The Real-Time Workshop 5.0 (Release 13)	Allows installation of Real-Time Workshop Embedded Coder.

If you experience installation difficulties and have Web access, connect to the MathWorks home page (`http://www.mathworks.com`). Look for the Installation Troubleshooting Wizard in the Support section.

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, and user input	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Buttons and keys	**Boldface** with book title caps	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'whole')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace` font	MATLAB responds with
		`A =`
		`5`

Menu and dialog box titles	**Boldface** with book title caps	Choose the **File Options** menu.
New terms and for emphasis	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'method')`

1

Product Overview

This section contains the following topics:

Introduction

The Real-Time Workshop® Embedded Coder is a separate, add-on product for use with Real-Time Workshop.

The Real-Time Workshop Embedded Coder provides a framework for the development of production code that is optimized for speed, memory usage, and simplicity. The Real-Time Workshop Embedded Coder is intended for use in embedded systems.

Real-Time Workshop Embedded Coder generates code that is easy to read, trace, and customize for your production environment.

The Real-Time Workshop Embedded Coder generates code in the `Embedded-C` format. Optimizations inherent in the `Embedded-C` code format include

- Use of *real-time model* data structure optimizes memory usage specifically for your model. (In many cases, this structure can be removed entirely from the generated code.)
- Simplified calling interface reduces overhead and lets you easily incorporate the generated code into hand-written application code. Model output and update functions are combined into a single routine.
- In-lined S-functions (required) reduce calling overhead and code size.
- Static memory allocation reduces overhead and promotes deterministic performance.

The Real-Time Workshop Embedded Coder supports the following key features:

- Automatic generation of an example main program, with comments detailing how to deploy the code generated for a model with or without an operating system
- Automatic generation of a deterministic multirate scheduler for single- and multitasking environments (further simplifying code deployment)
- Single- and multiple instance code generation; both using static memory allocation
- Supports asynchronous interrupt-driven execution of models with either single or multiple sample rates
- Integer only code generation

- Floating-point code generation (ANSI or ISO C library calls supported)
- Automatic generation of S-function wrappers, allowing you to validate the generated code in Simulink (Software-in-the-loop)
- Detailed HTML report fully documents the generated code, including active hyperlinks that trace code segments back to the model. The report describes code modules and helps to identify code generation optimizations relevant to your program.
- Code generation options let you optimize performance of data initialization and reduce ROM usage.
- Custom storage classes give you precise control over data symbols in the generated code, allowing you to interface virtually any class of structured or unstructured data.
- Automatic generation of an ASAP2 data export file to interface with commercial automotive calibration systems.
- Full support for all features of Simulink external mode. (See the "External Mode" section of the Real-Time Workshop documentation.)

This document describes the components of the Real-Time Workshop Embedded Coder provided with Real-Time Workshop. It also describes options for optimizing your generated code, and for automatically generating an S-function wrapper that calls your Real-Time Workshop Embedded Coder generated code from Simulink. In addition, certain restrictions that apply to the use of the Real-Time Workshop Embedded Coder are discussed.

We assume you have read "Program Architecture" and "Models with Multiple Sample Rates" in the Real-Time Workshop documentation. Those sections give a general overview of the architecture and execution of programs generated by Real-Time Workshop.

Real-Time Workshop Embedded Coder Demos and Examples

Demos

We have provided a number of demos to help you become familiar with features of the Real-Time Workshop Embedded Coder and to inspect generated code. These demos illustrate features specific to the Real-Time Workshop Embedded Coder as well as general Real-Time Workshop features as used with the Embedded Coder.

If you are reading this document online in the MATLAB Help browser, you can run the demos by clicking on the links in the **Command** column of the following table.

Alternatively, you can access the demo suite by typing commands from the **Command** column of the table at the MATLAB command prompt, as in this example:

```
ecoderdemos
```

Table 1-1: Real-Time Workshop Embedded Coder Demos

Command	Demo Topic
`ecoderdemos`	Top-level demo containing buttons to run the other demos of the Real-Time Workshop Embedded Coder demo suite
`ecodertutorial`	Interactive tutorial on application deployment, configuring options, custom target development, and backwards compatibility issues.
`asap2demo`	ASAP2 data file generation
`atomicdemo`	Nonvirtual subsystem code generation
`cbdemo`	High-level optimizations in generated code
`cscdemos`	Code generation with custom storage classes.
`cscdesignintro`	How to design your own custom storage classes.
`cscpredefineddemo`	Use of the predefined custom storage classes

Table 1-1: Real-Time Workshop Embedded Coder Demos (Continued)

Command	Demo Topic
`cscvariantdemo`	Use of variant parameters
`cscgetsetdemo`	Use of the `GetSet` custom storage class with data store memory blocks
`ecdemo`	Generation of callable procedure with pure integer code; also creates HTML code generation report
`ecifdemo`	Use of control flow constructs such as if, while, and for
`exprfolding`	Expression folding: a technique that improves code efficiency by reducing use of temporary variables and expressions
`hierdemo`	Resolution of variable names within a model hierarchy
`objectdemo`	Use of Simulink data objects in simulation and code generation
`rtwecexamplemain`	Generating an example main program for a bare-board target without an operating system.
`sfexfold`	Expression folding in a model that integrates Stateflow and Embedded Coder
`ssdemo`	Advanced features of Embedded Coder, including subsystem code generation, HTML code generation report, and automatic S-function wrapper generation
`tunabledemo`	Use of tunable expressions in generated code

ECRobot Target Example

The ECRobot (Embedded Coder Robot) target is a simple example of a custom target based on the Real-Time Workshop Embedded Coder.

Programs generated by the ECRobot target run on the Robot Command System (RCX™)[1] module of the LEGO® MINDSTORMS™ Robotics Invention System2.0™.

1. MINDSTORMS, RCX, Robotics Invention System 2.0, and LEGO are registered trademarks of The LEGO Group.

This platform affords an inexpensive and simple way to study concepts and techniques essential to developing a custom embedded target, and to develop, run and observe generated programs. The files included with the target illustrate typical approaches to problems encountered in custom target development, including

- Interfacing a Real-Time Workshop Embedded Coder generated program to an external real-time operating system (RTOS) or kernel.
- Implementing device drivers, via wrapper S-functions, for use in simulation and inlined code generation.
- Customizing a system target file by adding code generation options and adding the target to the System Target File Browser.
- Customizing a template makefile to use a target specific cross-compiler and download generated code to the target hardware.

The ECRobot target, originally developed as a training class example and demonstration, is now available to all Real-Time Workshop Embedded Coder users. The ECRobot target files are automatically installed with the Real-Time Workshop Embedded Coder. Source code files, control files, demonstration models, and documentation for the target are installed in the directory *matlabroot*`/toolbox/rtw/targets/ECRobot`.

Note The ECRobot target requires an operating system kernel, a cross-compiler and support utilities that are available on the Web. For instructions on how to obtain and install these utilities, see the file `readme.html` in the *matlabroot*`/toolbox/rtw/targets/ECRobot/documentation` directory.

2

Data Structures and Program Execution

This section describes the main data structures of the code generated by Real-Time Workshop Embedded Coder. It also summarizes the code modules and header files that make up a Real-Time Workshop Embedded Coder program, and describes where to find them. In addition, this section describes how Real-Time Workshop Embedded Coder generated programs execute, from the top level down to timer interrupt level. This section contains the following topics:

Data Structures and Code Modules

Real-Time Model Data Structure

The Real-Time Workshop Embedded Coder encapsulates information about the root model in the *real-time model* data structure. We refer to the real-time model data structure as `rtM`.

To reduce memory requirements, `rtM` 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. `rtM` may also contain model-specific `rtM` information related to timing, solvers, and model data such as inputs, outputs, states, and parameters.

By default, `rtM` 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 will further reduce memory usage. Selecting this option may also cause `rtM` to disappear completely from the generated code.

The symbol definitions for `rtM` 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`):

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

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

```
extern RT_MODEL_model *model_M;
```

Accessor Macros

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

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

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

To interface your code to `rtM`s of more than one model, simply include the * 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 rtM */
  errStatus = rtmGetErrorStatus(modelA_M);
  /* Refer to model B s rtM */
  errStatus = rtmGetErrorStatus(modelB_M);
}
```

Table 2-1 summarizes the `rtM` error status macros. To view other `rtM` related macros that are applicable to your specific model, generate code with a code generation report (See “Generating a Code Generation Report” on page 3–10); then view *`model`*`.h` via the hyperlink in the report.

Table 2-1: rtM Error Status Macros

Macro	Argument(s)	Return Type	Description
`rtmGetErrorStatus(rtm)`	`rtm`: reference to real-time model struct	`char *`	Returns most recent error status string.
`rtmSetErrorStatus(rtm,val)`	`rtm`: reference to real-time model struct `val`: C string	N/A	Set error status field of real-time model struct to the string `val`.

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 3–10 for further information.

Generated Code Modules

The 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`.

Table 2-2 summarizes the structure of source code generated by the Real-Time Workshop Embedded Coder.

Note The file packaging of the 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.

Table 2-2: Real-Time Workshop Embedded Coder File Packaging

File	Description
model.c	Contains entry points for all code implementing the model algorithm (*model*_step, *model*_initialize, *model*_terminate, *model*_SetEventsForThisBaseStep).
*model*_private.h	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 *model*_private.h when interfacing hand-written code to a model.
model.h	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 (*model*_rtM) via accessor macros. *model*.h is included by subsystem .c files in the model. If you are interfacing your hand-written code to generated code for one or more models, you should include *model*.h for each model to which you want to interface.
*model*_data.c (conditional)	*model*_data.c is conditionally generated. It contains the declarations for the parameters data structure and the constant block I/O data structure. If these data structures are not used in the model, *model*_data.c is not generated. Note that these structures are declared extern in *model*.h.
*model*_types.h	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. *model*_types.h is included by all the generated header files in the model.

Table 2-2: Real-Time Workshop Embedded Coder File Packaging (Continued)

File	Description
`ert_main.c` (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 2-7.
`autobuild.h` (optional)	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 2-7.) `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. See "The Static Main Program Module" on page 2-23 for further information.
`model_pt.c` (optional)	Provides data structures that enable a running program to access model parameters without use of external mode. To learn how to generate and use the `model_pt.c` file, see "C API for Parameter Tuning" in the Real-Time Workshop documentation.
`model_bio.c` (optional)	Provides data structures that enable your code to access block outputs. To learn how to generate and use the `model_bio.c` file, see "Signal Monitoring via Block Outputs" in the Real-Time Workshop documentation.

Note You can also control generation of code at the subsystem level, for any nonvirtual subsystem.You can instruct Real-Time Workshop to generate separate functions, within separate code files, for any nonvirtual subsystems. You can control both the names of the functions and of the code files generated from nonvirtual subsystems. See "Nonvirtual Subsystem Code Generation" in the Real-Time Workshop documentation for further information. Also, you can use custom storage classes to partition generated data structures. See "Custom Storage Classes" on page 4-1 for further information.

User-Written Code Modules

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

We recommend that you establish a working directory for your own code modules. Your working directory should be on the MATLAB path. You must also modify the Real-Time Workshop Embedded Coder template makefile and system target file so that the build process can find your source and object files. See "Targeting Real-Time Systems" in the Real-Time Workshop documentation for information.

Generating the Main Program

The **Generate an example main program** option controls whether or not `ert_main.c` is generated. This option is located in the `ERT code generation options (3)` category of the Real-Time Workshop pane of the **Simulation Parameters** dialog box, as shown in this figure.

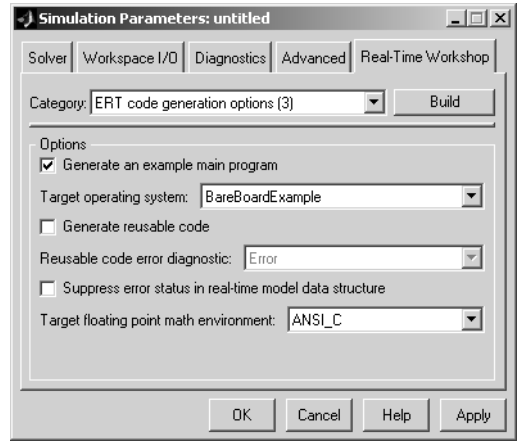


Figure 2-1: Options for Generating a Main Program

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` 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 2-9.

If you turn the **Generate an example main program** option off, the Real-Time Workshop Embedded Coder provides the module `ert_main.c` as a basis for your custom modifications (see "The Static Main Program Module" on page 2-23).

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.

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 2-10 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 2-18 describes the operation of example programs designed for deployment under the VxWorks real-time operating system.
- "Model Entry Points" on page 2-20 describes the model functions that are generated for both stand-alone and VxWorks example programs.

Stand-Alone Program Execution

By default, the Real-Time Workshop Embedded Coder generates self-sufficient programs that do not require an external real-time executive or operating system. We refer to such programs as *stand-alone* programs. 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 box, select the **Generate an example main program** option (this option is on by default).

2. When **Generate an example main program** is selected, the **Target operating system** pop-up menu is enabled. Select `BareBoardExample` from this menu (this option is the default selection).

The core of a stand-alone program is the main loop. On each iteration, the main loop executes a background or null task and checks for a termination condition.

The main loop is periodically interrupted by a timer. The Real-Time Workshop function `rt_OneStep` is either installed as a timer interrupt service routine (ISR), or called from a timer ISR at each clock step.

The execution driver, `rt_OneStep`, sequences calls to the *`model`*`_step` function. 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.

If your model includes device driver blocks, the *`model`*`_step` function will incorporate your inlined driver code to perform I/O functions such as reading inputs from an analog-digital converter (ADC) or writing computed outputs to a digital-analog converter (DAC).

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 interupts
  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` 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` 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 `rtM` 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`)

Table 2-3 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.

Table 2-3: 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. Since by definition the only legal solver mode for a single-rate model is `SingleTasking`, we refer to this case simply as "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
  ModelStep-- Time step combines output,logging,update
}
```

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 reinterruption will be detected through the overrun flag.

Reinterruption 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. The following pseudocode shows the design of `rt_OneStep` in a multi-rate multitasking program.

```
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 i=1:NumTasks    -- iterate over sub-rate tasks
    Check for sub-rate interrupt overflow
    If (sub-rate task i is scheduled)
      ModelStep(tid=i)    --sub-rate time step
    EndIf
  EndFor
}
```

In a multi-rate multitasking system, the Real-Time Workshop Embedded Coder uses a prioritized, preemptive multitasking scheme to execute the different sample rates in your model.

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.

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.

On each invocation, `rt_OneStep` makes one or more calls to *`model`*`_step`, passing in the appropriate `tid`. The `tid` informs *`model`*`_step` that all blocks having that `tid` should execute. `rt_OneStep` always calls *`model`*`_step(tid = 0)` because the base-rate task must execute on every clock step.

On each clock tick, `rt_OneStep` and *`model`*`_step` maintain scheduling counters and *event flags* for each sub-rate task. Both the counters and 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`*`_step`. 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 via 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.

After updating its scheduling data structures and stepping the base-rate task, `rt_OneStep` iterates over the scheduling flags in `tid` order, calling *`model`*`_step(tid)` for any task whose flag is set. This ensures that tasks are executed in order of priority.

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 will preempt lower priority tasks. Upon return from interrupt, lower priority tasks will resume in the previously scheduled order.

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 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).

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. The only task is the base-rate task. On each clock tick, `rt_OneStep` checks the overrun flag and calls *`model`*`_step`, passing in `tid 0`.

The generated *`model`*`_step` code maintains scheduling counters on each clock tick, via the `rate_monotonic_scheduler` function. There is one counter for each sample rate in the model. The counters are implemented as an array (`model_M.cTaskTicks[]`) within `rtM`.

The counters are, in effect, clock rate dividers that count up the sample period associated with each sample rate in the model. When a counter indicates that a sample period for a given rate has elapsed, `rate_monotonic_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, using macros provided for the purpose (`rtmIsSampleHit` and `rtmIsSpecialSampleHit`).

Guidelines for Modifying rt_OneStep

`rt_OneStep` does not require extensive modification. The only required modification is to reenable 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_step(0)`.
- Get model outputs associated with the base rate after calling `model_step(0)`.
- Set model inputs associated with sub-rates before calling `model_step(tid)` in the sub-rate loop.
- Get model outputs associated with sub-rates after calling `model_step(tid)` 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 `rtM`) 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. We strongly recommend that you 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 2–10, 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 box, select the **Generate an example main program** option (this option is on by default).
2. When **Generate an example main program** is selected, the **Target operating system** pop-up menu is enabled. Select `VxWorksExample` from this menu.

Some modifications to the generated code are required; comments in the generated code identify the required modifications.

Task Management

In a VxWorks example program, the main program and the base rate and sub-rate tasks (if any) run as proritized 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 Table 2-3). 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. The base rate task calls *model*_step with tid 0, while the sub-rate tasks call *model*_step with their associated tids. The base rate task and *model*_step are responsible for maintaining event flags and scheduling counter, 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_monotonic_scheduler utility, which maintains the scheduling counters that determine which rates should execute. *model*_step is responsible for checking the counters, using macros provided for the purpose (rtmIsSampleHit and rtmIsSpecialSampleHit).

Model Entry Points

This section discusses the entry points to the generated code.

Note carefully that the calling interface generated for each of these functions will differ significantly depending on how you set the **Generate reusable code** option (See "Generate Reusable Code" on page 3-18).

By default, **Generate reusable code** is off, and the model entry point functions access model data via 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 via *`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. In a single-rate model, the *`model`*`_step` function prototype is

```
void model_step(void);
```

In a multi-rate model, the *`model`*`_step` function prototype is

```
void model_step(int_T tid);
```

where `tid` is a task identifier. The `tid` is determined by logic within `rt_OneStep` (See "rt_OneStep" on page 2-12).

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.

Single-Rate Operation. In a single-rate model, *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.

Multi-Rate Operation. In a multi-rate model, *model*_step execution is almost identical to single-rate execution, except for the use of the task identifier (tid) argument.

The caller (rt_OneStep) assigns each block a tid (See "rt_OneStep" on page 2-12). *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 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 rtM 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 "Basic Code Generation Options" on page 3-3 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 2-19 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` module (see "Generating the Main Program" on page 2-7).

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. `ert_main.c` 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 `ert_main.c`, you may need to continue using this module.

When developing applications using `ert_main.c`, we recommend that you copy `ert_main.c` 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 will create *`model`*`_ert_main.obj` (on Unix, *`model`*`_ert_main.o`) in the build directory.

`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.

In the static version of `ert_main.c`, the operation of `rt_OneStep` and the main function are essentially the same as described in "Stand-Alone Program Execution" on page 2-10.

Modifying the Static Main Program

As in a generated program, a few modifications to the main loop and `rt_OneStep` are necessary. See "Guidelines for Modifying the Main Program" on page 2-11 and "Guidelines for Modifying rt_OneStep" on page 2-16.

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, note that multi-rate systems will 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, the 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 will raise 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 will raise a compile-time error if **External Mode** is used illegally.

  ```
  #ifdef EXT_MODE
  ```

3

Code Generation Options and Optimizations

This section contains the following topics:

Controlling and Optimizing the Generated Code

The Real-Time Workshop Embedded Coder features a number of code generation options that can help you further optimize the generated code. The Real-Time Workshop Embedded Coder can also produce a code generation report in HTML format. This report documents code modules and helps you to identify optimizations that are relevant to your model.

"Basic Code Generation Options" on page 3-3 documents code generation options you can use to improve performance and reduce code size.

"Controlling Stack Space Allocation" on page 3-8 discusses options related to the storage of signals.

Please see "Optimizing the Model for Code Generation" in the Real-Time Workshop documentation for information about code optimization techniques common to all code formats.

Basic Code Generation Options

To access the basic code generation options, select the **Real-Time Workshop** tab of the **Simulation Parameters** dialog box. Then select `ERT code generation options (1)` from the **Category** menu.

Figure 3-1 displays the basic code generation options (with default settings) for the Real-Time Workshop Embedded Coder.

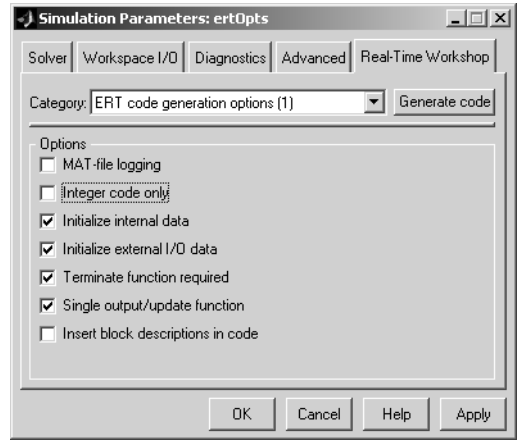


Figure 3-1: Basic Code Generation Options

Setting the basic code generation options as follows will result in more highly optimized code:

- Consider clearing the **Initialize internal data** and **Initialize external I/O data** options.

  These options (both on 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 **Initialize internal data** redundant.

  However, be aware that if **Initialize internal data** is turned off, it is not guaranteed that memory will be in a known state each time the generated code begins execution. If you turn the option off, 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 off **Initialize internal data** if you want to test the behavior of your design during a warm boot (i.e., a restart without full system reinitializiation).

In cases where you have turned off **Initialize internal data** 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, **Initialize floats and doubles to 0.0**, lets you control the representation of zero used during initialization. See "Initialize Floats and Doubles to 0.0" on page 3–17.

Note that the code still initializes data structures whose value is not zero when **Initialize internal data** and **Initialize external I/O data** are selected.

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

- Clear the **Terminate function required** option if you do not require a terminate function for your model.
- Select the **Single output/update function** check box. Combining the output and update functions is the default. This option generates the *`model`*`_step` call, which reduces overhead and allows Real-Time Workshop to use more local variables in the step function of the model.
- If your application uses only integer arithmetic, select the **Integer code only** option to ensure that generated code contains no floating-point data or operations. When this option is selected, an error is raised if any noninteger data or expressions are encountered during code generation. The error message reports the offending blocks and parameters.
- Clear the **MAT-file logging** option. 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 has the following effects:

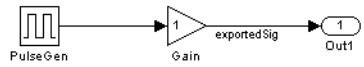
- Under certain conditions, code and storage associated with root output ports are eliminated, achieving further efficiency. See "Virtualized Output Ports Optimization" on page 3-6 for information.
- The *model*_step function does not check the current time against the stop time. Therefore the generated program runs indefinitely, regardless of the setting of the model's stop time. The ert_main program displays a message notifying the user that the program will run indefinitely.

Virtualized Output Ports Optimization

The *virtualized output ports* optimization lets you eliminate code and data storage associated with root output ports under the following conditions:

1. The **MAT-file logging** option is cleared (this is the default for the Embedded Coder).
2. The TLC variable `FullRootOutputVector` equals `0`. This is the default for the Embedded Coder.
3. The signal line entering the root output port is stored as a global variable. (See the "Code Generation and the Build Process" chapter of the Real-Time Workshop documentation for information on how to control signal storage in generated code.)

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



In the default case (conditions 1 and 2 above are true), the output of the Gain block is written to the signal storage location, `exportedSig`. No code or data is generated for the `Out1` block, which has become, in effect, a virtual block. This is shown in the following code fragment.

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

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

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

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

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

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

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

```
%assign FullRootOutputVector = 1
```

to the Embedded Coder system target file. Alternatively, you can enter the assignment into the **System Target File** field on the Real-Time Workshop pane of the **Simulation Parameters** dialog box.

Generating Code from Subsystems

When generating code from a subsystem, we recommend that you set the sample times of all subsystem inports explicitly.

Generating Block Comments

When the **Insert block descriptions in code** option is selected, comments are inserted into the code generated for any blocks that have text in their **Description** fields.

To generate block comments:

1. Right-click on the block you want to comment. Select **Block Properties** from the context menu. The **Block Properties** dialog box opens.
2. Type the comment into the **Description** field.
3. Select the **Insert block descriptions in code** option in the `ERT code generation options (1)` category of the Real-Time Workshop pane.

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

Controlling Stack Space Allocation

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 (i.e., in stack variables).
- Control the allocation of stack space when using local storage.

For a complete discussion of signal storage options, see the "Code Generation and the Build Process" chapter of the Real-Time Workshop documentation.

If you want to store signals in stack space, you must turn the **Local block outputs** option on. To do this:

1. Select the **Advanced** tab of the **Simulation Parameters** dialog box. Make sure that the **Signal storage reuse** is on. If **Signal storage reuse** is off, the **Local block outputs** option is not available.
2. Click **Apply** if necessary.
3. Select the **Real-Time Workshop** tab of the **Simulation Parameters** dialog box.
4. From the **Category** menu, select `General code generation options`.
5. Select the **Local block outputs** option. Click **Apply** if necessary.

Your embedded application may be constrained by limited stack space. When the **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 functions in the entire model may not exceed `MaxStackSize` (in bytes). `MaxStackSize` can be any positive integer. If the total size of local variables

exceeds this maximum, the Target Language Compiler will allocate the remaining variables in global, rather than local, memory.

The default value for `MaxStackSize` is `rtInf`, i.e., unlimited stack size.

- `MaxStackVariableSize`: Limits the size of any local variable declared in a function to `N` bytes, where `N>0`. A variable whose size exceeds `MaxStackVariableSize` will be allocated in global, rather than local, memory.

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

```
%assign MaxStackSize = 4096
```

We recommend that you 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 documentation.

Generating a Code Generation Report

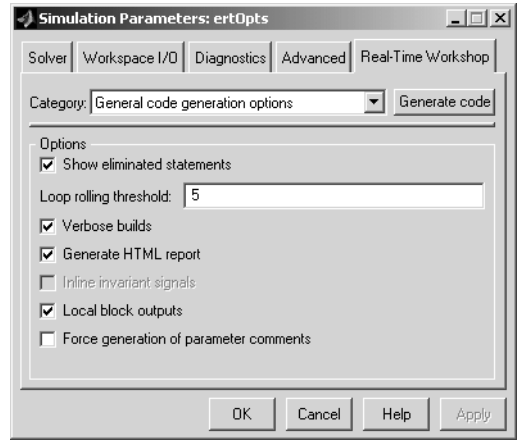
The Real-Time Workshop Embedded Coder code generation report is an enhanced version of the HTML code generation report normally generated by Real-Time Workshop. The report consists of several sections:

- 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 the MATLAB Help browser. 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 will 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. Select the **Real-Time Workshop** tab of the **Simulation Parameters** dialog box. Then select `General code generation options` from the **Category** menu.

2. Select **Generate HTML report**, as shown in this picture.



3 Follow the usual procedure for generating code from your model or subsystem.

4 Real-Time Workshop writes the code generation report file in the build directory. The file is named *`model`*`_codegen_rpt.html` or *`subsystem`*`_codegen_rpt.html`.

5 Real-Time Workshop automatically opens the MATLAB Help browser and displays the code generation report.

Alternatively, you can view the code generation report in your Web browser.

Automatic S-Function Wrapper Generation

An S-function wrapper is an S-function that calls your 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 documentation.

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

- A noninlined 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 code validation and 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`, in the build directory. This module contains the S-function that calls the Real-Time Workshop Embedded Coder code that you deploy. This S-function can be used within Simulink.

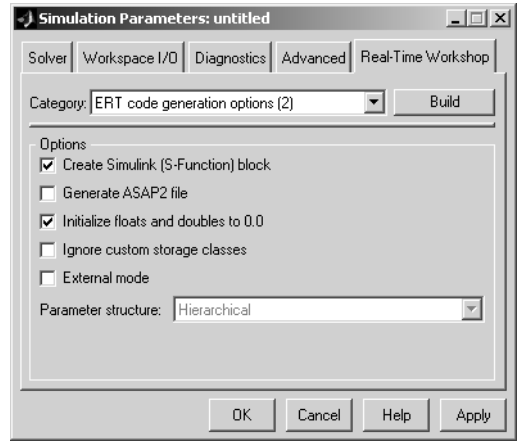
The build process then compiles and links *model*`_sf.c` with *model*`.c` and the other Real-Time Workshop Embedded Coder generated code modules, building a MEX-file. The MEX-file is named *model*`_sf.`*mexext*. (*mexext* is the file extension for MEX-files on your platform, as given by the MATLAB `mexext` 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.

Generating an S-Function Wrapper

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

1. Select the **Real-Time Workshop** tab of the **Simulation Parameters** dialog box. Then select `ERT code generation options (2)` from the **Category** menu.

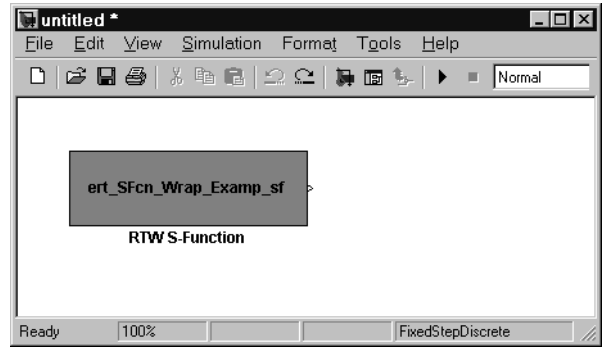
2. Select the **Create Simulink (S-Function) block** option, as shown.



3 Configure the other code generation options as required.

4 Click the **Build** button.

5 When the build process completes, an untitled model window opens. This model contains the generated S-Function block.



6 Save the new model.

7 The generated S-Function block is now ready to use with other blocks or models in Simulink.

Limitations

It is not possible to create multiple instances of a Real-Time Workshop Embedded Coder generated S-Function block within a model, because the code uses static memory allocation.

Other Code Generation Options

This section describes advanced Real-Time Workshop Embedded Coder code generation options. These options are found in the `ERT code generation options (2)` and `ERT code generation options (3)` items of the **Category** menu of the **Real-Time Workshop** tab of the **Simulation Parameters** dialog box. Figure 3-2 and Figure 3-3 display the advanced code generation options for the Real-Time Workshop Embedded Coder.

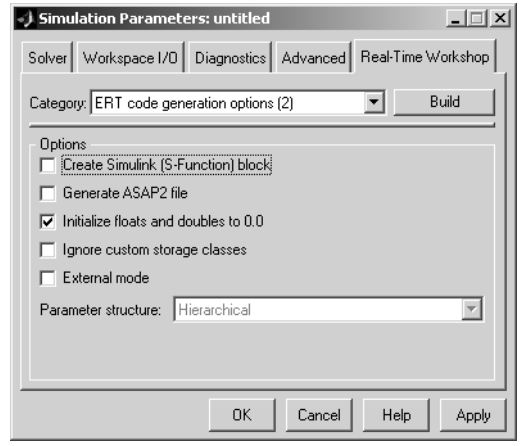


Figure 3-2: ERT Code Generation Options (2)

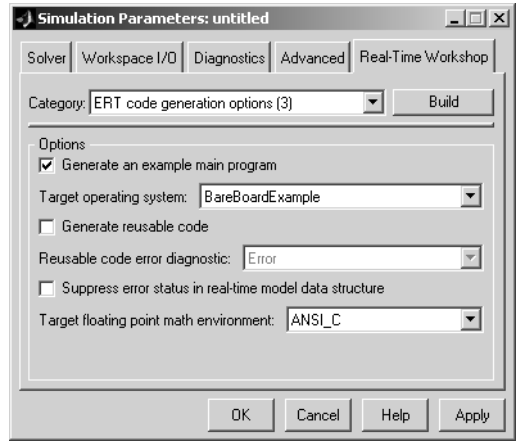


Figure 3-3: ERT Code Generation Options (3)

The following sections describe these options, in order of their appearance on the dialog box.

Create Simulink (S-Function) Block

See “Generating an S-Function Wrapper” on page 3-12 for information on this feature.

Generate ASAP2 File

The Real-Time Workshop Embedded Coder **Generate ASAP2 File** code generation option lets you export an ASAP2 file containing information about your model during the code generation process.

The ASAP2 file generation process requires information about your model's parameters and signals. Some of this information is contained in the model itself. The rest must be supplied by using Simulink data objects with the necessary properties. Simulink provides two data classes to assist you in providing the necessary information. See “Generating ASAP2 Files” on page A-1 for information on this feature.

Initialize Floats and Doubles to 0.0

This option lets you control how internal storage for floats and doubles is initialized. This option affects code generation only if you have turned on the **Initialize internal data** option (see "Basic Code Generation Options" on page 3-3).

When this option is off, all internal storage (regardless of type) is cleared to the integer bit pattern `0` (that is, all bits are off).

When this option is on (the default), additional code is generated to set float and double storage explicitly to the value `0.0`. This additional code 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 using the default (off).

Ignore Custom Storage Classes

When this option is selected, objects with custom storage classes are treated as if their storage class attribute is set to `Auto`. By default this option is off for the Real-Time Workshop Embedded Coder. See "Introduction to Custom Storage Classes" on page 4-2 for further information.

External Mode

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

If you are unfamiliar with external mode, we recommend that you start with the external mode exercise in the "Quick Start Tutorials" section of the Real-Time Workshop documentation.

Like the GRT, GRT malloc, and Tornado targets, the Real-Time Workshop Embedded Coder supports host/target communication via TCP/IP, using the standard `ext_comm` MEX-file. If you need to support external mode on a custom target using your own low-level communications layer, see the "Targeting Real-Time Systems" section of the Real-Time Workshop documentation for detailed information on the external mode API.

Parameter Structure

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 for further information.)

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, the 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** pop-up 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, the Real-Time Workshop Embedded Coder generates a single parameter data structure. This is a flat data structure; subsystem parameters are defined as fields within this structure.

Generate An Example Main Program

This option and the related **Target operating system** pop-up menu let you generate a model-specific example main program module. See "Generating the Main Program" on page 2-7.

Generate Reusable Code

The **Generate reusable code** option and the related **Reusable code error diagnostic** menu let you generate reusable, reentrant code from a model or subsystem. When this option is selected, data structures such as block states, parameters, external outputs, etc. are passed in (by reference) as arguments to

*model*_step and other generated model functions. These data structures are also exported via *model*.h.

In some cases, the Real-Time Workshop Embedded Coder may generate code that will compile but is not reentrant. For example, if any signal, DWork structure, or parameter data has a storage class other than Auto, global data structures will be 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.
- Warn: build proceeds after displaying a warning message.
- Error: build aborts after displaying an error message.

In some cases, the 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.

- A Stateflow chart that outputs function-call events
- An S-function that is not code-reuse compliant
- A subsystem triggered by a wide function call trigger

In these cases, the build will terminate after reporting the problem.

When **Generate reusable code** option is not selected (the default), model data structures are statically allocated and accessed directly in the model code. Therefore the model code is neither reusable nor reentrant.

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 (rtM) includes an error status field (data type string). This field lets you log and monitor error messages via macros provided for this purpose (see *model*.h). The error status field is initialized to NULL. If **Suppress error status in real-time model data structure** is selected, the error status field is not included in rtM. 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 will be 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.

Target Floating Point Math Environment

This pop-up menu provides two options. If you select the `ANSI_C` option (the default), the Real-Time Workshop Embedded Coder generates calls to the ANSI C (ANSI X3.159-1989) math library for floating-point functions. If you select the `ISO_C` option, Real-Time Workshop Embedded Coder generates calls to the ISO C (ISO/IEC 9899:1999) math library wherever possible.

If your target compiler supports the ISO C (ISO/IEC 9899:1999) math library, we recommend selecting the `ISO_C` option and setting your compiler's ISO C option. This will generate 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.

If your target compiler does not support ISO C math library functions, use the `ANSI_C` option.

4

Custom Storage Classes

This section contains the following topics:

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 "Code Generation and the Build Process" chapter of the Real-Time Workshop documentation.

The built-in storage classes are suitable for a simulation or rapid prototyping environment, but embedded system designers often require greater control over the representation of data. For example, you may need to

- Conserve memory by storing Boolean data in bit fields.
- Integrate the code generated by Real Time Workshop with legacy software whose interfaces cannot be modified.
- Employ certain constructs to comply with your organization's software engineering guidelines for safety-critical code.

The Real-Time Workshop Embedded Coder's *custom storage classes* provide extended control over the constructs required to represent data in an embedded algorithm. A custom storage class is defined by a set of Target Language Compiler (TLC) instructions that the Real Time Workshop uses when generating code for each type of reference to data of that class. These instructions tell the Real Time Workshop exactly how to define, declare, and access the data. Since the instructions are created by the user, the variations in the code generated are unlimited.

The Real Time Workshop Embedded Coder includes a set of predefined custom storage classes designed to be useful in embedded systems development. You can use these classes without any TLC programming. The sections that follow

explain the custom storage classes provided and supported by The MathWorks for use with the Real Time Workshop Embedded Coder.

The TLC code for each predefined storage class is found in *matlabroot*`/toolbox/simulink/simulink/@Simulink/tlc`. If you want to create your own custom storage classes, you can use this code as an example. However, the creation of new classes is outside the scope of this document.

Properties of Predefined Custom Storage Classes

The Real-Time Workshop Embedded Coder defines two classes of custom data objects:

- `Simulink.CustomParameter`: 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 "Predefined Custom Storage Class Summaries" on page 4-4. 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 "Class-Specific Storage Class Attributes" on page 4-8.
  - `Value`. This property is the numeric value of the object, used as an initial (or inlined) parameter value in generated code.
- `Simulink.CustomSignal`: 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 "Predefined Custom Storage Class Summaries" 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 "Class-Specific Storage Class Attributes" on page 4-8.

Predefined Custom Storage Class Summaries

The following tables summarize the predefined custom storage classes. 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 Table 4-1, insert type qualifiers in the data declaration.

Table 4-1: Const, ConstVolatile, and Volatile Storage Classes

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 Table 4-2, handles issues of data scope and file partitioning.

Table 4-2: ExportToFile, ImportFromFile, and Internal Storage Classes

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
`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 Table 4-3, specify the data structure or construct used to represent the data.

Table 4-3: BitField, Define, and Struct Storage Classes

Class Name	Purpose	Parameters	Signals	Data types	Complex	Wide
`BitField`	Embed Boolean data in a named bit field	Y	Y	Boolean	N	N

Table 4-3: BitField, Define, and Struct Storage Classes (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

Table 4-4: Data Access Storage Classes

Class Name	Purpose	Parameters	Signals	Data types	Complex	Wide
`GetSet`	Read and write data using access functions. See "GetSet Custom Storage Class for Data Store Memory" on page 4-10	N	Y	any	N	Y

Class-Specific Storage Class Attributes

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`).

Table 4-5 summarizes the storage classes with additional attributes, and the meaning of those attributes. Attributes marked optional have default values and may be left unassigned.

Table 4-5: Additional Properties of Custom Storage Classes

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 *`model`*`_export.h` by default.	Y
`ImportFromFile`	`FileName`	String. Defines the name of the generated header file which to be used in `#include` directive.	N
`ImportFromFile`	`IncludeDelimeter`	Enumerated. Defines delimiter used for filename in the `#include` directive. Delimiter is either double quotes (e.g. `#include "vars.h"`) or angle brackets (e.g. `#include <vars.h>`). The default is quotes.	Y
`BitField`	`BitFieldName`	String. Defines name of bit field in which data will be embedded; if unassigned, the name defaults to `rt_BitField`.	Y

Table 4-5: Additional Properties of Custom Storage Classes (Continued)

Storage Class Name	Additional Properties	Description	Optional (has default)
`Struct`	`StructName`	String. Defines name of the struct in which data will be embedded; if unassigned, the name defaults to `rt_Struct`.	Y
`GetSet`	`GetFunction`	String. Specifies function call to read data. See "GetSet Custom Storage Class for Data Store Memory" on page 4-10.	
`GetSet`	`SetFunction`	String. Specifies function call to write data. See "GetSet Custom Storage Class for Data Store Memory" on page 4-10.	
`GetSet`	`HeaderFile`	String. Same as `FileName` in the `ImportFromFile` class.	
`GetSet`	`IncludeDelimeter`	Enumerated. Same as `IncludeDelimeter` in the `ImportFromFile` class.	

Other Custom Storage Classes

This section discusses other custom storage classes that have been provided for special purposes.

GetSet Custom Storage Class for Data Store Memory

The `GetSet` custom storage class can only be used for the memory of Data Store Read and Data Store Write blocks. The properties of the `GetSet` class are summarized in Table 4-4 and Table 4-5.

This class supports only signals of noncomplex data types. Its purpose is to generate code that reads (gets) and writes (sets) data via functions. For example, if the `GetFunction` for signal `x` is specified as `"get_x"` then the generated code will call `get_x()` wherever the value of `x` is used. If the `SetFunction` for signal `x` is specified as `"set_x"` then the generated code will call `set_x(value)` wherever the value of `x` is assigned.

For wide signals, an additional index argument is passed, as in

```
get_x(idx)
set_x(value, idx)
```

The `cscgetsetdemo` demo illustrates the use of the `GetSet` custom storage class.

Designing Custom Storage Classes

Designing your own custom storage classes is an advanced topic. We have provided a step-by-step tutorial with the Real-Time Workshop Embedded Coder demo suite. To view this tutorial, type the following command at the MATLAB command prompt:

```
cscdesignintro
```

Assigning a Custom Storage Class to Data

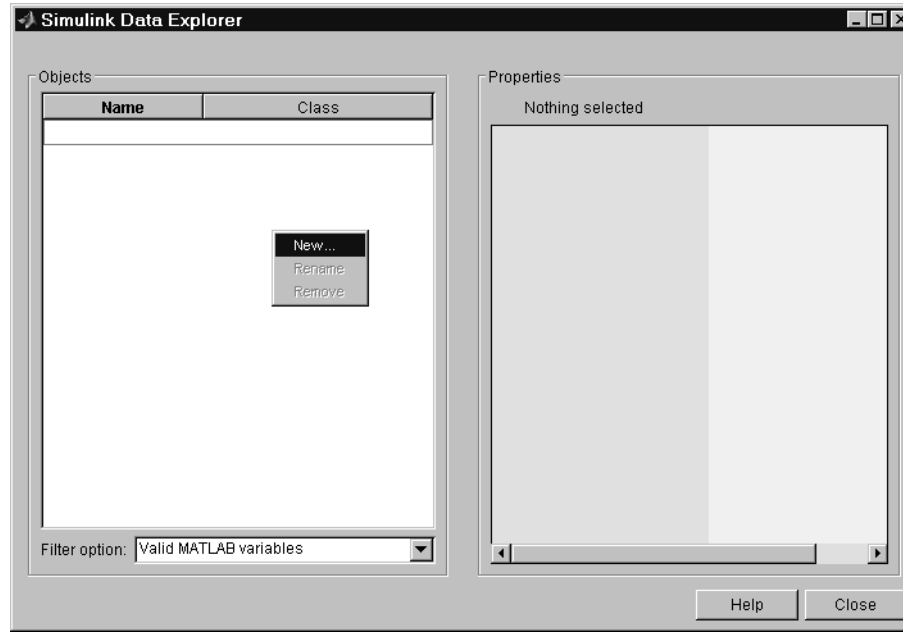
You can assign a custom storage class to a Simulink data object either from the Simulink Data Explorer, or from the MATLAB command prompt.

Assigning a Custom Storage Classes via the Simulink Data Explorer

To create a custom parameter or signal object from the Simulink Data Explorer:

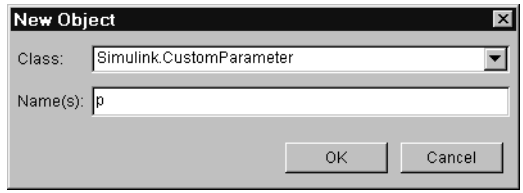
1. Choose **Data explorer** from the Simulink **Tools** menu, or type `slexplr` at the MATLAB prompt. The **Data explorer** dialog box appears.

2. In the **Objects** (left) pane, depress the right mouse button. A pop-up menu appears. Select **New...** from the menu, as shown in this figure.



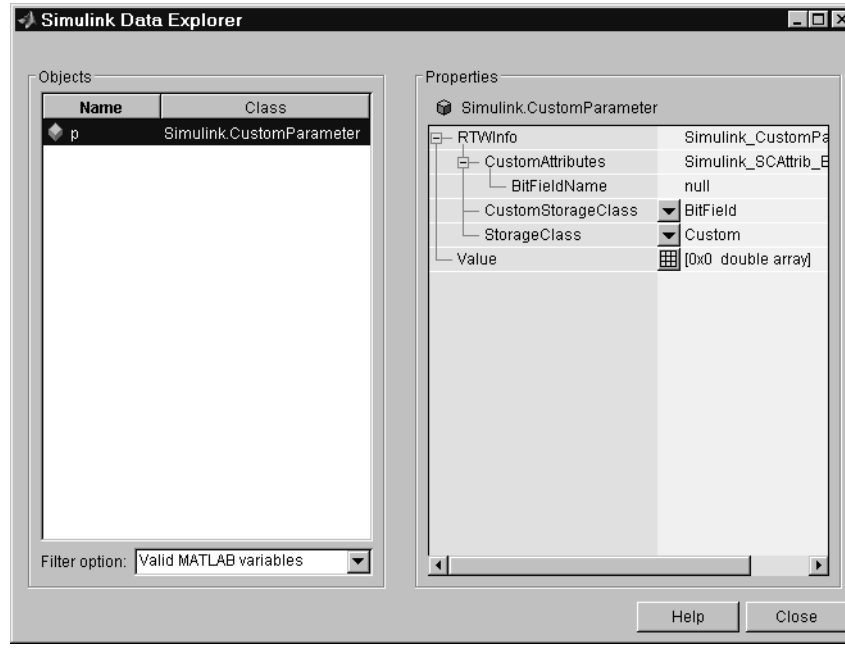
3 A **New Object** dialog box is displayed. Select **Simulink.CustomParameter** or **Simulink.CustomSignal** from the **Class** menu. Enter the name of the object in the **Name** field.

In this figure, an object `p` of class `Simulink.CustomParameter` is created.

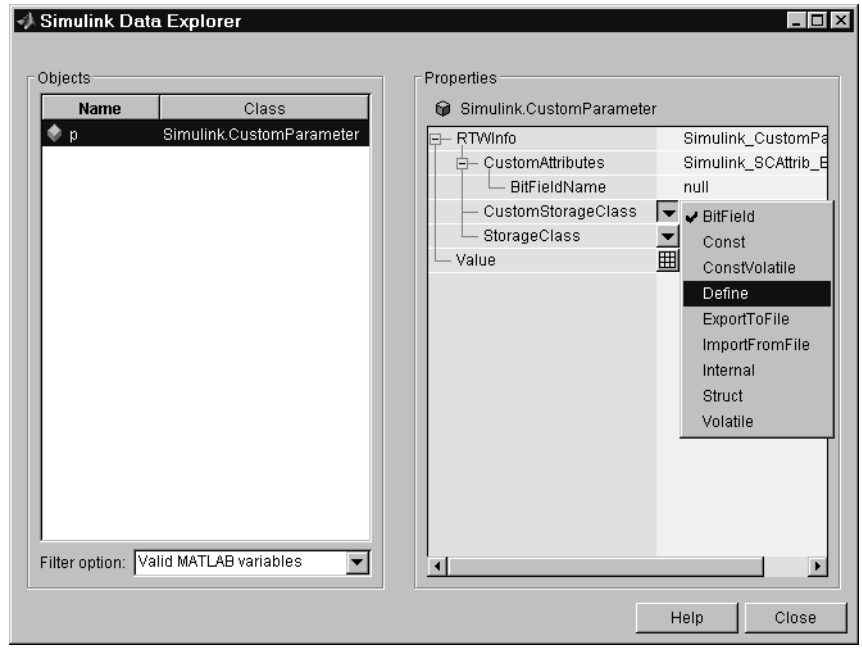


4 Click **OK**. The object now appears in the **Objects** pane, and is selected. Its properties are shown in the **Properties** (right) pane.

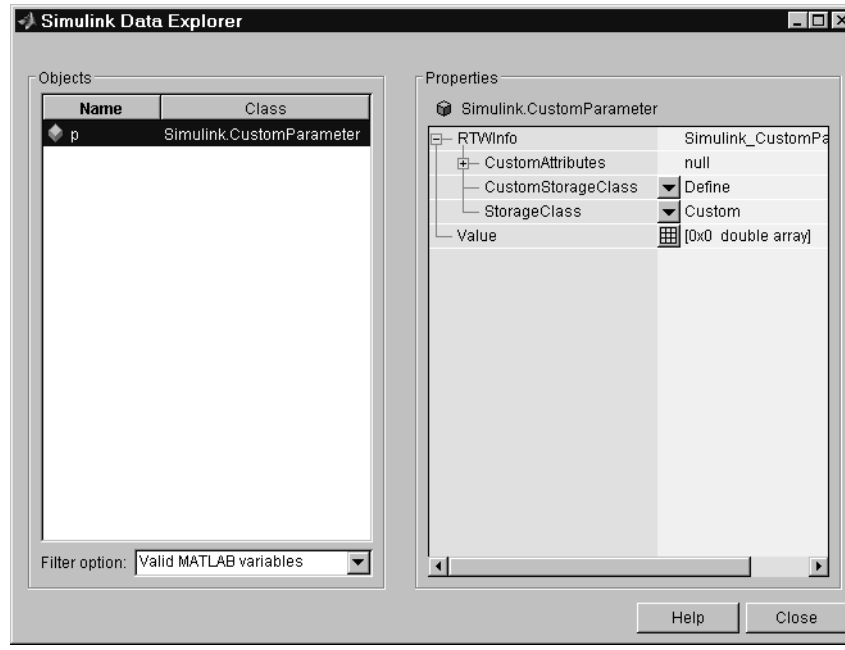
5 To view the properties of the object, open the `RTWInfo` field by clicking on the + button next to the field name. Similarly, open the `RTWInfo.CustomAttributes` field to show the class-specific attributes of the object (if any). This figure shows the initial properties of the `Simulink.CustomParameter` object `p`.



6 Set the custom storage class of the object, by selecting a value for the object's `RTWInfo.CustomStorageClass` property from the **CustomStorageClass** menu. In this figure, the custom storage class of the object `p` is being changed from **BitField** to **Define**.



7 This figure shows the object properties after the custom storage class property of `p` is set to `Define`. Notice that the `BitFieldName` attribute is no longer displayed, since that attribute applies only to objects whose custom storage class is `BitField`.



8 Make sure that the `RTWInfo.StorageClass` property is set to `Custom`. If this property is not set to `Custom`, the custom storage properties are ignored.

9 Click **Close** to dismiss the Simulink Data Explorer.

Assigning a Custom Storage Class via the MATLAB Command Line

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 (optional) `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 its default value, `Custom`. If you inadvertently set this property to some other value, the custom storage properties are ignored.

Code Generation with 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, please see the discussion of Simulink data objects in the "Code Generation and the Build Process" chapter of the Real-Time Workshop documentation.

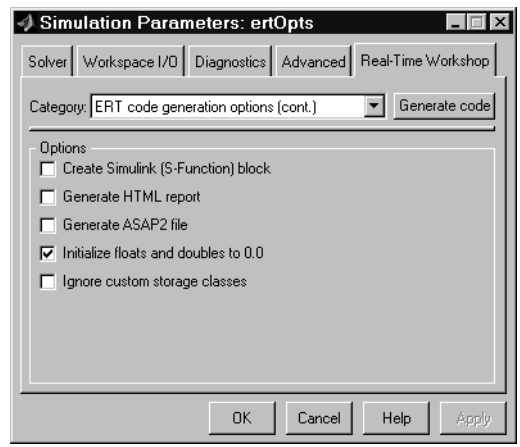
To generate code with custom storage classes, you must

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, as shown in this picture. This is the default for the 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 rapid prototyping target such as the generic real-time target (GRT), without having to reconfigure your parameter and signal objects.

When using the Real-Time Workshop Embedded Coder, you can control the **Ignore custom storage classes** option via the check box in the **ERT code generation options (2)** category of the Real-Time Workshop tab of the **Simulation Parameters** dialog box.

If you are using a target that does not have a check box for this option (such as a custom target) you can enter the option directly into the **System target file** field in the **Target configuration** category of the Real-Time Workshop pane. The following example turns the option on:

```
-aIgnoreCustomStorageClasses=1
```

Ordering of Generated Storage Declarations

Variables, structs, and other declarations in the generated code are sorted

1. Alphabetically by storage class
2. Within storage class, alphabetically by variable name

See the code excerpts in the next section, “Sample Code Excerpts” for examples of how declarations are sorted.

Sample Code Excerpts

In the model shown in Figure 4-1, block parameters and signals are associated with data objects belonging to each of the predefined custom storage classes, as follows:

- Parameters `c`, `cv`, and `d` reference `Simulink.CustomParameter` objects with custom storage class `Const`, `ConstVolatile` and `Define`, respectively.
- Signals `v` and `intl` reference `Simulink.CustomSignal` objects with custom storage class `Volatile` and `Internal`, respectively.
- Signals `sw1` and `sw2` reference `Simulink.CustomSignal` objects with custom storage class `Struct`, whose `StructName` storage class attribute is set to `testpoints`.
- Signals `b1` and `b2` reference `Simulink.CustomSignal` objects with custom storage class `BitField`, whose `BitFieldName` storage class attribute is set to `signalBit`.
- Parameter `eg` references a `Simulink.CustomParameter` object with custom storage class `ExportToFile`, whose `FileName` storage class attribute is set to `exportedSignals.h`.
- Parameter `ig` references a `Simulink.CustomParameter` object with custom storage class `ImportFromFile`, whose `FileName` storage class attribute is set to `importedSignals.h`, and whose `IncludeDelimeter` attribute is set to `Braces`.

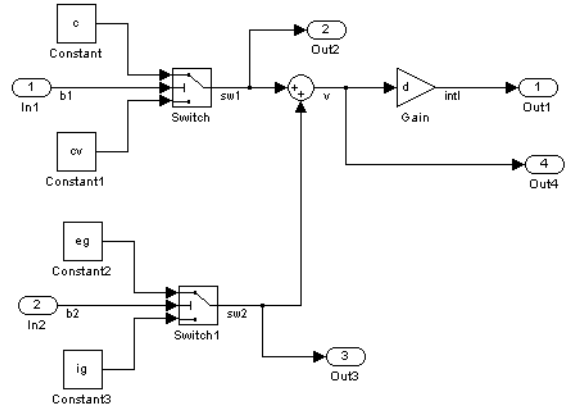


Figure 4-1: Model Using Custom Storage Classes

The structure definitions for the bit field `signalBit` and the struct `testpoints` are in the generated file *`model`*`_types.h`, as shown in the following code excerpt. Notice also the inclusion of the generated file `exportedSignals.h` and the file `importedSignals.h`. The latter is assumed to be a hand-written file containing external signal definitions:

```
#include "exportedSignals.h"
#include <importedSignals.h>

typedef struct signalBit_tag {
  unsigned int b1:1;
  unsigned int b2:1;
} signalBit_bitfield;

/* Struct data */
typedef struct testpoints_tag {
  real_T sw1;
  real_T sw2;
} testpoints_struct;
```

A code excerpt from `exportedSignals.h` follows, declaring the parameter `eg` and making it visible to externally written code:

```
#ifndef _exportedSignals_h
  #include "tmwtypes.h"
  extern real_T eg;
  #define _exportedSignals_h
#endif
```

The following excerpt from the generated file *`model`*`_data.c` contains the storage declarations and initializers for the parameters `c`, `cv`, and `d`; signals `v` and `intl`; exported signal `eg`; signals `b1` and `b2` (embedded in `signalBit`); and `sw1` and `sw2` (embedded in `testpoints`):

```
/* Data with custom storage class Const */
const real_T c = 2.0;

/* Data with custom storage class ConstVolatile */
const volatile real_T cv = 4.0;

/* Data with custom storage class Define */
#define d                               5.0

/* Data with custom storage class ExportToFile */
real_T eg;

/* Data with custom storage class Internal */
real_T intl;

/* Data with custom storage class Volatile */
volatile real_T v;

/* External Outputs Structure */
ExternalOutputs rtY;

/* user code (bottom of parameter file) */
signalBit_bitfield signalBit = {0,0};
testpoints_struct testpoints = {0.0,0.0};
```

The following code excerpt from *model*.c illustrates the application of these variables in the generated program:

```
/* Switch: '<Root>/Switch' incorporates:
 *   Inport: '<Root>/In1'
 *   Constant: '<Root>/Constant'
 *   Constant: '<Root>/Constant1'
 */
if (signalBit.b1) {
  testpoints.sw1 = c;
} else {
  testpoints.sw1 = cv;
}

/* Switch: '<Root>/Switch1' incorporates:
 *   Inport: '<Root>/In2'
 *   Constant: '<Root>/Constant2'
 *   Constant: '<Root>/Constant3'
 */
if (signalBit.b2) {
  testpoints.sw2 = eg;
} else {
  testpoints.sw2 = ig;
}

/* Sum: '<Root>/Sum' */
v = testpoints.sw1 + testpoints.sw2;

/* Gain: '<Root>/Gain'
 *
 * Regarding '<Root>/Gain':
 *   Gain value: d
 */
intl = v * d;

/* Outport: '<Root>/Out1' */
rtY.Out1 = intl;
```

5

Requirements, Restrictions, Target Files

This section contains the following topics:

Requirements and Restrictions

- By definition, a Real-Time Workshop Embedded Coder program operates in discrete time. Your model must use the following solver options:
  - Solver type: fixed-step
  - Algorithm: discrete (no continuous states)
- You must select the `SingleTasking` or `Auto` solver mode when the model is single-rate. Table 2-3, Permitted Solver Modes for Real-Time Workshop Embedded Coder Targeted Models, indicates permitted solver modes for single-rate and multirate models.
- You cannot have any continuous time blocks in your model (see "Unsupported Blocks" on page 5-2).
- If you are designing a program that is intended to run indefinitely, you should not use blocks that have a dependency on absolute time. See "Blocks that Depend on Absolute Time" in the Real-Time Workshop documentation for a list of such blocks.
- You must inline all S-functions with a corresponding TLC file. The reason for this is that Real-Time Workshop Embedded Coder generated code uses the real-time object, rather than the `SimStruct`. Since noninlined S-functions require reference to the `SimStruct`, they cannot be used in Real-Time Workshop Embedded Coder generated programs. See the Simulink Writing S-Functions documentation for information about inlining S-functions.

Unsupported Blocks

The `Embedded-C` format does not support the following built-in blocks:

- Continuous
  - No blocks in this library are supported
- Discrete
  - First-Order Hold
- Functions and Tables
  - MATLAB Fcn
  - The following S-functions: M-file and Fortran S-functions, or noninlined C-MEX S-functions that call into MATLAB
- Math

  - Algebraic Constraint
- Nonlinear
  - Rate Limiter
- Sources
  - Clock
  - Chirp Signal
  - Ramp
  - Repeating Sequence
  - Signal Generator

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_lcc.tmf` — LCC compiler
- `ert_unix.tmf` — UNIX host
- `ert_vc.tmf` — Visual C
- `ert_watc.tmf` — Watcom C

A

Generating ASAP2 Files

ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a standard description you use for data measurement, calibration, and diagnostic systems.

This section includes the following topics:

Overview

Real-Time Workshop lets you export an ASAP2 file containing information about your model during the code generation process.

To make use of ASAP2 file generation, you should become familiar with the following topics:

- ASAM and the ASAP2 standard and terminology. See the ASAM Web site at `http://www.asam.de`.
- Simulink data objects. Data objects are used to supply information not contained in the model. For an overview, see “Working with Data Objects” in the Using Simulink documentation.
- Storage and representation of signals and parameters in generated code. See “Working with Data Structures” in the Real-Time Workshop documentation.
- Signal and parameter objects and their use in code generation. See “Working with Data Structures” in the Real-Time Workshop documentation.

If you are reading this document online in the MATLAB Help browser, you can run an interactive demo of ASAP2 file generation.

Alternatively, you can access the demo by typing the following command at the MATLAB command prompt, as in this example:

```
asap2demo
```

Targets Supporting ASAP2

Real-Time Workshop provides two target configurations you can use to generate ASAP2 files. You can select either of these target configurations from the System Target File Browser:

- The **ASAM-ASAP2 Data Definition Target** lets you generate only an ASAP2 file, without building an executable.
- The **Real-Time Windows Embedded Coder** lets you generate an ASAP2 file as part of the code generation and build process.

Procedures for generating ASAP2 files via these targets are given in "Generating an ASAP2 File" on page A-6.

Alternatively, you can add ASAP2 support to your own target by defining the TLC variable `GenerateASAP2` in your system target file, as shown in the following code example:

```
%assign GenerateASAP2 = 1
%include "codegenentry.tlc"
```

Note You must define `GenerateASAP2` before including `codegenentry.tlc`.

Defining ASAP2 Information

The ASAP2 file generation process requires information about your model's parameters and signals. Some of this information is contained in the model itself. The rest must be supplied by using Simulink data objects with the necessary properties.

Real-Time Workshop provides two example data classes to assist you in providing the necessary information. The classes are

- `ASAP2.Parameter`, a subclass of `Simulink.Parameter`
- `ASAP2.Signal`, a subclass of `Simulink.Signal`

This document refers to these as the *ASAP2 classes*, and to objects instantiated from these classes as *ASAP2 objects*. The ASAP2 class creation files are located in the directory *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2`. To create ASAP2 objects, make sure that this directory is on the MATLAB path.

As with the built-in `Simulink.Parameter` and `Simulink.Signal` classes, we recommend that you create your own packages and classes rather than using the ASAP2 classes directly. To do this, copy and rename the directory *`matlabroot`*`/toolbox/rtw/targets/asap2/asap2/@ASAP2`, and modify the class creation files it contains. You can extend the ASAP2 classes if additional properties are required. For general information about extending data object classes, see “Working with Data Objects” in the Using Simulink documentation.

The following table contains the minimum set of data attributes required for ASAP2 file generation. Some data attributes are defined in the model; others are supplied in the properties of ASAP2 objects. For attributes that are defined in `ASAP2.Parameter` or `ASAP2.Signal` objects, the table gives the associated property name.

Table A-1: Data Required for ASAP2 File Generation

Data Attribute	Defined In	Property Name
Data type	Model	Not applicable
Scaling (if fixed point data type)	Model	Not applicable
Name (Symbol)	Data object	Inherited from name of handle to the data object to which parameter or signal name resolves
Long identifier (Description)	Data object	`LongID_ASAP2`
Minimum allowable value	Data object	`PhysicalMin_ASAP2`
Maximum allowable value	Data object	`PhysicalMax_ASAP2`
Units	Data object	`Units_ASAP2`
Memory Address (optional)	Data object (see note below)	`MemoryAddress_ASAP2` (optional; see "Memory Address Attribute" below)

Memory Address Attribute. The Memory Address attribute, if known before code generation, can be defined in the data object. Otherwise, a placeholder string is inserted. You can replace the placeholder with the actual address by post-processing the generated file. See the file
`matlabroot/toolbox/rtw/targets/asap2/asap2/asap2post.m`
for an example.

Generating an ASAP2 File

You can generate an ASAP2 file from your model in one of the following ways:

- Use the Real-Time Windows Embedded Coder to generate an ASAP2 file as part of the code generation and build process.
- Use the ASAM-ASAP2 Data Definition Target to generate only an ASAP2 file, without building an executable.
- Add ASAP2 support to your own target (see "Targets Supporting ASAP2" on page A-3).

This section discusses how to generate an ASAP2 file via the targets that have built-in ASAP2 support.

Generating ASAP2 Files via the Real-Time Windows Embedded Coder

The procedure for generating a mode's data definition in ASAP2 format via the Real-Time Windows Embedded Coder is as follows:

1. Create the desired model. Use appropriate parameter names and signal labels to refer to `CHARACTERISTICS` and `MEASUREMENTS` respectively.

2. Define the relevant `ASAP2.Parameter` and `ASAP2.Signal` objects in the MATLAB workspace.

3. Configure the data objects to generate unstructured global storage declarations in the generated code by assigning one of the following storage classes to the `RTWInfo.StorageClass` property:

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

4. Configure the other data object properties such as `LongID_ASAP2`, `PhysicalMin_ASAP2`, etc.

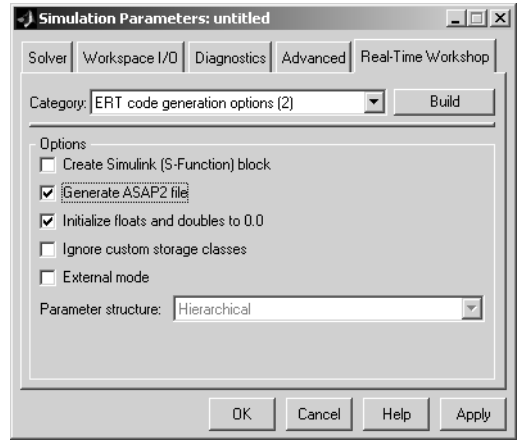
5. In the Advanced pane of the **Simulation Parameters** dialog box, select the **Inline parameters** option.

   Note that you should *not* configure the parameters associated with your data objects in the **Model Parameter Configuration** dialog box. If a parameter

that resolves to a Simulink data object is configured using the **Model Parameter Configuration** dialog box, the dialog box configuration is ignored. You can, however, use the **Model Parameter Configuration** dialog to configure other parameters in your model.

6 In the Real-Time Workshop pane, click **Browse** to open the System Target File Browser. In the browser, select the **Real-Time Windows Embedded Coder Target**.

7 Select `ERT code generation options (2)` from the **Category** menu of the Real-Time Workshop pane. Then select the **Generate ASAP2 file** option.



8 Click **Apply**.

9 Click **Build** (or **Generate code**).

10 Real-Time Workshop writes the ASAP2 file to the build directory. The ASAP2 filename is controlled by the ASAP2 setup file. By default, the file is named *`model`*`.a2l`.

Generating ASAP2 Files via the ASAM-ASAP2 Data Definition Target

The procedure for generating a model's data definition in ASAP2 format via the ASAM-ASAP2 Data Definition Target is as follows:

1. Create the desired model. Use appropriate parameter names and signal labels to refer to `CHARACTERISTICS` and `MEASUREMENTS` respectively.

2. Define the relevant `ASAP2.Parameter` and `ASAP2.Signal` objects in the MATLAB workspace.

3. Configure the data objects to generate unstructured global storage declarations in the generated code by assigning one of the following storage classes to the `RTWInfo.StorageClass` property:

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

4. Configure the other data object properties such as `LongID_ASAP2`, `PhysicalMin_ASAP2`, etc.

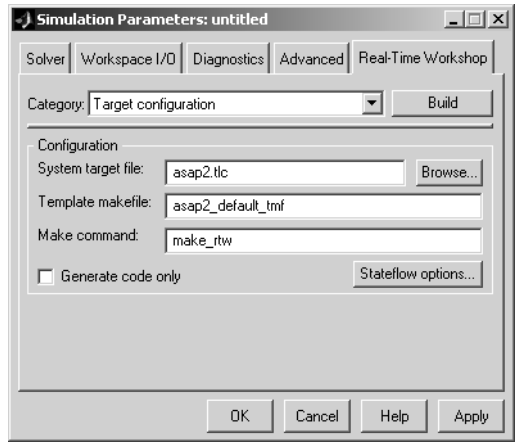
5. In the Advanced pane of the **Simulation Parameters** dialog box, select the **Inline parameters** option.

   Note that you should *not* configure the parameters associated with your data objects in the **Model Parameter Configuration** dialog box. If a parameter that resolves to a Simulink data object is configured using the **Model Parameter Configuration** dialog box, the dialog box configuration is ignored. You can, however, use the **Model Parameter Configuration** dialog to configure other parameters in your model.

6. In the Real-Time Workshop pane, click **Browse** to open the System Target File Browser. In the browser, select the **ASAM-ASAP2 Data Definition Target**.

7 Select `Target configuration` from the **Category** menu of the Real-Time Workshop pane. Then select the **Generate code only** option.

This picture shows the correct configuration.



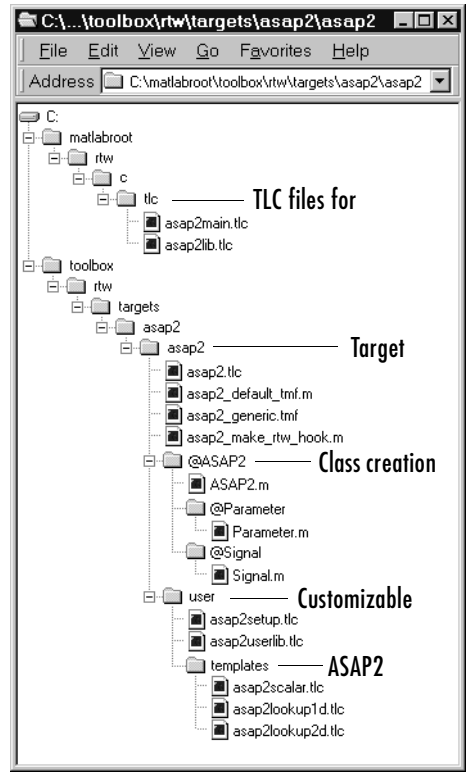
8 Click **Apply**.

9 Click **Generate code**.

10 Real-Time Workshop writes the ASAP2 file to the build directory. The ASAP2 filename is controlled by the ASAP2 setup file. By default, the file is named *`model`*`.a2l`.

Customizing an ASAP2 File

The Real-Time Workshop Embedded Coder provides a number of TLC files to enable you to customize the ASAP2 file generated from a Simulink model. The following figure illustrates the hierarchy of ASAP2 related directories and files within the MATLAB directory.



ASAP2 File Structure on the MATLAB Path

The ASAP2 related files are located within the directories shown above. The files are organized as follows:

- TLC files for generating ASAP2 files

  The *matlabroot*/rtw/c/tlc directory contains TLC files that generate ASAP2 files. These files are included by the Real-Time Workshop Embedded Coder and ASAP2 system target files (ert.tlc and asap2.tlc).

- ASAP2 target files

  The *matlabroot*/toolbox/rtw/targets/asap2/asap2 directory contains the ASAP2 system target file and other control files.

- ASAP2 class creation files

  The *matlabroot*/toolbox/rtw/targets/asap2/asap2/@ASAP2 directory contains the M-files that define the ASAP2.Parameter and ASAP2.Signal classes.

- Customizable TLC files

  The *matlabroot*/toolbox/rtw/targets/asap2/asap2/user directory contains files that you can modify to customize the content of your ASAP2 files.

- ASAP2 templates

  The *matlabroot*/toolbox/rtw/targets/asap2/asap2/user/templates directory contains templates that define each type of CHARACTERISTIC in the ASAP2 file.

Customizing the Contents of the ASAP2 File

The ASAP2 related TLC files enable you to customize the appearance of the ASAP2 file generated from a Simulink model. Most customization is done by modifying or adding to the files contained in the *matlabroot*/toolbox/rtw/targets/asap2/asap2/user directory.This section refers to this directory as the asap2/user directory.

The user-customizable files provided are divided into two groups:

- The *static* files define the parts of the ASAP2 file that are related to the environment in which the generated code is used. They describe information specific to the user and/or project. The static files are not model-dependant.
- The *dynamic* files define the parts of the ASAP2 file that are generated based on the structure of the source model.

The procedure for customizing the ASAP2 file is as follows:

1 Make a copy of the `asap2/user` directory before making any modifications.

2 Remove the old `asap2/user` directory from the MATLAB path, or add the new `asap2/user` directory to the MATLAB path above the old directory. This will ensure that MATLAB uses the new ASAP2 setup file, `asap2setup.tlc`.

   `asap2setup.tlc` specifies which directories and files to include in the TLC path during the ASAP2 file generation process. Modify `asap2setup.tlc` to control the directories and folders included in the TLC path.

3 Modify the static parts of the ASAP2 file. These include
   - Project and header symbols, which are specified in `asap2setup.tlc`
   - Static sections of the file, such as file header and tail, `A2ML`, `MOD_COMMON`, etc. These are specified in `asap2userlib.tlc`.
   - Specify the appearance of the dynamic contents of the ASAP2 file by modifying the existing ASAP2 templates, or by defining new ASAP2 templates. Sections of the ASAP2 file affected include
   - `RECORD_LAYOUTS`: modify appropriate parts of the ASAP2 template files.
   - `CHARACTERISTICS`: modify appropriate parts of the ASAP2 template files.

     For more information on modifying the appearance of `CHARACTERISTICS`, see "ASAP2 Templates" on page A-12.
   - `MEASUREMENTS`: These are specified in `asap2userlib.tlc`.
   - `COMPU_METHODS`: These are specified in `asap2userlib.tlc`.

ASAP2 Templates

The appearance of `CHARACTERISTICS` in the ASAP2 file is controlled using a different template for each type of `CHARACTERISTIC`. The `asap2/user` directory contains template definition files for scalars, 1-D Lookup Table blocks and 2-D Lookup Table blocks. You can modify these template definition files, or you can create additional templates as required.

The procedure for creating a new ASAP2 template is as follows:

1 Define a parameter group. See "Defining Parameter Groups" on page A-13.

2 Create a template definition file. See “Creating Template Definition Files” on page A-14.

3 Include the template definition file in the TLC path. The path is specified in the ASAP2 setup file, `asap2setup.tlc`.

Defining Parameter Groups. In some cases you must group multiple parameters together in the ASAP2 file (for example, the `x` and `y` data in a 1-D Lookup Table block). Parameter groups enable Simulink blocks to define an associative relationship between some or all of their parameters. The following example shows the `Lookup1D` parameter group and describes how to create and use parameter groups in conjunction with the ASAP2 file generation process.

The `BlockInstanceSetup` function, within a block's TLC file, creates parameter groups. There are two built-in TLC functions that facilitate this process: `SLibCreateParameterGroup` and `SLibAddMember`. The following code fragment creates the `Lookup1D` parameter group in `look_up.tlc`. Similar syntax is used to create parameter groups for the Look-Up Table (2D) block, the Fixed-Point Blockset Look-Up Table block, and the Look-Up Table (2-D) block:

```
%if GenerateInterfaceAPI
  %% Create a parameter group for ASAP2 data definition
  %assign group  = SLibCreateParameterGroup(block,"Lookup1D")
  %assign tmpVar = SLibAddMember(block,group,InputValues)
  %assign tmpVar = SLibAddMember(block,group,OutputValues)
%endif
```

ParameterGroup records are not written to the *model*.rtw file, but are included as part of the relevant Block records in the CompiledModel. The following code fragment shows the Lookup1D parameter group. The Lookup1D parameter group has two Member records. The Reference fields of these records refer to the relevant x and y data records in the GlobalMemoryMap:

```
Block {
  Type       Lookup
  Name       "<Root>/Look-Up Table"
  ...
  NumParameterGroups    1
  ParameterGroup {
    Name         Lookup1D
    NumMembers   2
    Member {
      NumMembers   0
      Reference    ...
    }
    Member {
      NumMembers   0
      Reference    ...
    }
  }
}
```

The Lookup1D parameter group is used by the function ASAP2UserFcnWriteCharacteristic_Lookup1D, which is defined in the template definition file, asap2lookup1d.tlc. This function uses the parameter group to obtain the references to the associated x and y data records in the GlobalMemoryMap, as shown in the following code fragment.

```
%function ASAP2UserFcnWriteCharacteristic_Lookup1D(paramGroup) Output
  %assign xParam = paramGroup.Member[0].Reference
  %assign yParam = paramGroup.Member[1].Reference
  ...
%endfunction
```

Creating Template Definition Files. This section describes the components that make up an ASAP2 template definition file. This description is in the form of code examples from asap2lookup1d.tlc, the template definition file for the

`Lookup1D` template. This template corresponds to the `Lookup1D` parameter group.

Note When creating a new template, use the corresponding parameter group name in place of `Lookup1D` in the code fragments shown.

The following sections describe the components of an ASAP2 template definition file.

Template Registration Function. The input argument is the name of the parameter group associated with this template:

```
%<LibASAP2RegisterTemplate("Lookup1D")>
```

RECORD_LAYOUT Name Definition Function. Record layout names (aliases) can be arbitrarily specified for each data type. This function is used by the other components of this file.

```
%function ASAP2UserFcnRecordLayoutAlias_Lookup1D(dtId) void
  %switch dtId
  %case tSS_UINT8
    %return "Lookup1D_UBYTE"
  ...
  %endswitch
%endfunction
```

Function to Write RECORD_LAYOUT Definitions. This function writes out `RECORD_LAYOUT` definitions associated with this template. The function is called by the built-in functions involved in the ASAP2 file generation process. The function name must be defined as shown, with the appropriate template name after the underscore:

```
%function ASAP2UserFcnWriteRecordLayout_Lookup1D() Output
  /begin RECORD_LAYOUT
%<ASAP2UserFcnRecordLayoutAlias_Lookup1D(tSS_UINT8)>
    ...
  /end   RECORD_LAYOUT
%endfunction
```

Function to Write the CHARACTERISTIC. This function writes out the `CHARACTERISTIC` associated with this template. The function is called by the built-in functions involved in the ASAP2 file generation process. The function name must be defined as shown, with the appropriate template name after the underscore.

The input argument to this function is a pointer to a parameter group record. The example shown is for a `Lookup1D` parameter group that has two members. The references to the associated `x` and `y` data records are obtained from the parameter group record as shown.

This function calls a number of built-in functions to obtain the required information. For example, `LibASAP2GetSymbol` returns the symbol (name) for the specified data record.

```
%function ASAP2UserFcnWriteCharacteristic_Lookup1D(paramGroup)
Output
  %assign xParam = paramGroup.Member[0].Reference
  %assign yParam = paramGroup.Member[1].Reference
  %assign dtId = LibASAP2GetDataTypeId(xParam)
    /begin CHARACTERISTIC
     /* Name */              %<LibASAP2GetSymbol(xParam)>
     /* Long identifier */   "%<LibASAP2GetLongID(xParam)>"
     ...
    /end CHARACTERISTIC
%endfunction
```

Structure of the ASAP2 File

Table A-2 outlines the basic structure of the ASAP2 file and describes which Target Language Compiler functions and files are used to create each part of the file:

- Static parts of the ASAP2 file are shown in **bold**.
- Function calls are indicated by `%<FunctionName()>`.

Table A-2: Sections of ASAP2 File and Related Target Language Compiler Functions and Files

File Section	Contents of asap2main.tlc	TLC File Containing Function Definition
File header	%<ASAP2UserFcnWriteFileHead()>	asap2userlib.tlc
/begin PROJECT ""	**/begin PROJECT** "%<ASAP2ProjectName>"	asap2setup.tlc
/begin HEADER "" HEADER contents **/end HEADER**	**/begin HEADER** "%<ASAP2HeaderName>" %<ASAP2UserFcnWriteHeader()> **/end HEADER**	asap2setup.tlc asap2userlib.tlc
/begin MODULE "" MODULE contents: - A2ML - MOD_PAR - MOD_COMMON ...	**/begin MODULE** "%<ASAP2ModuleName>" %<ASAP2UserFcnWriteHardwareInterface()>	asap2setup.tlc asap2userlib.tlc
Model-dependent MODULE contents: - RECORD_LAYOUTs - CHARACTERISTICS - ParameterGroups - ModelParameters	%<SLibASAP2WriteDynamicContents()> Calls user-defined functions: ...WriteRecordLayout_*TemplateName*() ...WriteCharacteristic_*TemplateName*() ...WriteCharacteristic_Scalar()	asap2lib.tlc user/templates/...
- MEASUREMENTS - ExternalInputs - BlockOutputs	...WriteMeasurement()	asap2userlib.tlc
- COMPU_METHODS	...WriteCompuMethod()	asap2userlib.tlc
/end MODULE	**/end MODULE**	
/end PROJECT	**/end PROJECT**	
File footer/tail	%<ASAP2UserFcnWriteFileTail()>	asap2userlib.tlc

Index

I

M

O

P

R

S

T

U

V