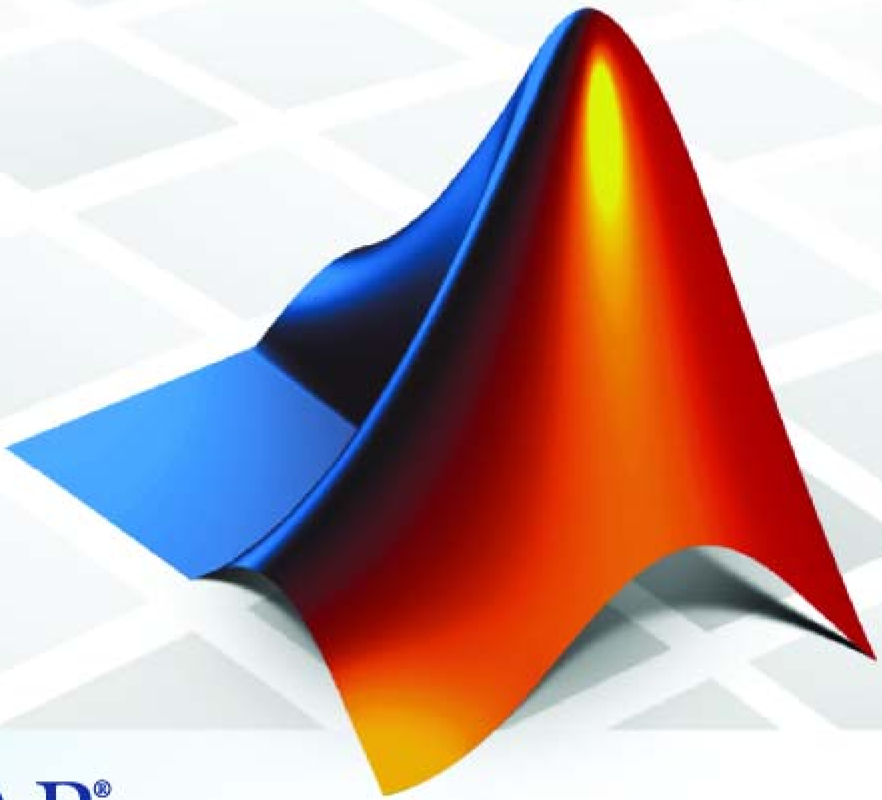
Real-Time Workshop® Embedded Coder 5
User's Guide



MATLAB® & SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

Real-Time Workshop Embedded Coder User's Guide

Revision History

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

Contents

1 Data Structures, Code Modules, and Program Execution

2 Code Generation Options and Optimizations

3 Custom Storage Classes

5 Advanced Code Generation Techniques

Data Structures, Code Modules, and Program Execution

This chapter describes the essential components and techniques used in a Real-Time Workshop Embedded Coder application. The topics include data structures, code modules, header files, generated program execution, and task management. For an introduction to Real-Time Workshop Embedded Coder and its capabilities, see *Getting Started with Real-Time Workshop Embedded Coder*.

Real-Time Model (rtModel) Data Structure

In this section...

Overview

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

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

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

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

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

  ```
  struct _RT_MODEL_model_Tag {
  ...
  };
  ```

- Forward declaration typedef (in *`model`*`_types.h`):

  ```
  typedef struct _RT_MODEL_model_Tag RT_MODEL_model;
  ```

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

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

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

  ```
  extern RT_MODEL_model *model_M;
  ```

rtModel Accessor Macros

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

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

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

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

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

void myHandWrittenFunction(void)
{
  const char_T *errStatus;

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

To view macros related to `rtModel` that are applicable to your specific model, generate code with a code generation report (see “Generating and Using an HTML Code Generation Report” on page 2-44). Then, view `model.h` by clicking the hyperlink in the report.

Code Modules

In this section...
"Introduction" on page 1-5
"Generated Code Modules" on page 1-5
"User-Written Code Modules" on page 1-8

Introduction

This section summarizes the code modules and header files that make up a Real-Time Workshop Embedded Coder program, and describes where to find them.

Note that in most cases, the easiest way to locate and examine the generated code files is to use the Real-Time Workshop Embedded Coder code generation report. The code generation report provides a table of hyperlinks that let you view the generated code in the MATLAB Help browser. See "Generating and Using an HTML Code Generation Report" on page 2-44 for further information.

Generated Code Modules

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

Real-Time Workshop Embedded Coder File Packaging on page 1-6 summarizes the structure of source code generated by Real-Time Workshop Embedded Coder.

Note The file packaging of Real-Time Workshop Embedded Coder differs slightly (but significantly) from the file packaging employed by the GRT, GRT malloc, and other non-embedded targets. See the Real-Time Workshop documentation for further information.

Real-Time Workshop Embedded Coder File Packaging

File	Description
`model.c` or `.cpp`	Contains entry points for all code implementing the model algorithm (for example, `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_M`) with accessor macros. `model.h` is included by subsystem `.c` or `.cpp` 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` or `.cpp` (conditional)	`model_data.c` or `.cpp` is conditionally generated. It contains the declarations for the parameters data structure, the constant block I/O data structure, and any zero representations used for the model's structure data types. If these data structures and zero representations are not used in the model, `model_data.c` or `.cpp` is not generated. Note that these structures and zero representations 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. Also provides type definitions for user-defined types used by the model.
`rtwtypes.h`	Defines data types, structures and macros required by Real-Time Workshop Embedded Coder generated code. Most other generated code modules require these definitions.
`ert_main.c` or `.cpp` (optional)	This file is generated only if the **Generate an example main program** option is on. (This option is on by default.) See "Generating the Main Program Module" on page 1-9.

Real-Time Workshop Embedded Coder File Packaging (Continued)

File	Description
autobuild.h (optional)	This file is generated only if the **Generate an example main program** option is off. (See "Generating the Main Program Module" on page 1-9.) 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 "Static Main Program Module" on page 1-26 for further information.
*model*_capi.c or .cpp *model*_capi.h (optional)	Provides data structures that enable a running program to access model parameters and signals without use of external mode. To learn how to generate and use the *model*_capi.c or .cpp and .h files, see the "Data Exchange APIs" chapter in the Real-Time Workshop documentation.

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

- Nonvirtual subsystem code generation: You can instruct Real-Time Workshop to generate separate functions, within separate code files, for any nonvirtual subsystems. You can control the names of the functions and of the code files. See "Nonvirtual Subsystem Code Generation" in the Real-Time Workshop documentation for further information.
- Custom storage classes: You can use custom storage classes to partition generated data structures into different files based on file names you specify. See Chapter 3, "Custom Storage Classes" for further information.
- Module Packaging Features (MPF) also lets you direct the generated code into a required set of .c or .cpp and .h files, and control the internal organization of the generated files. See the Module Packaging Features document for details.

User-Written Code Modules

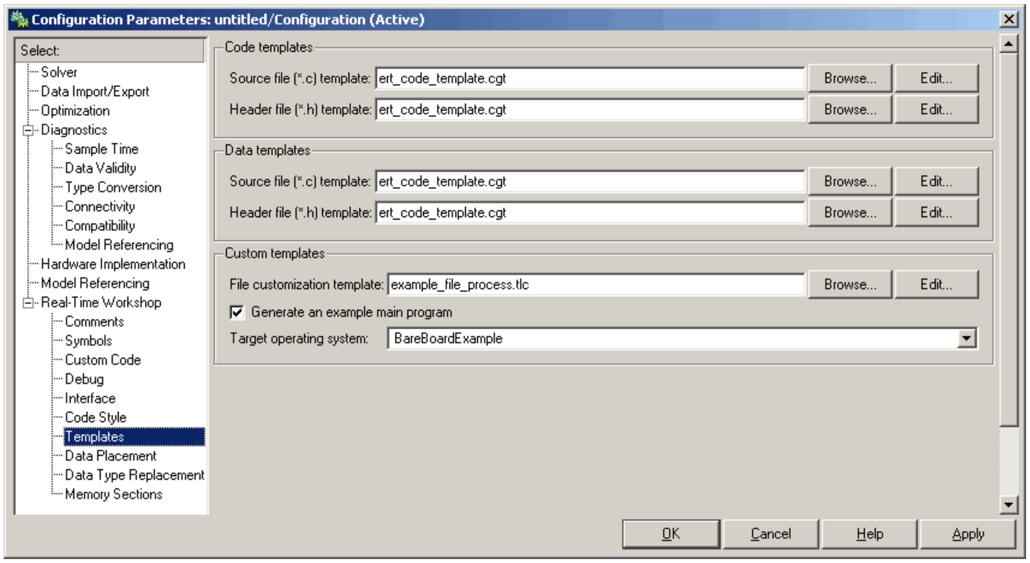
Code that you write to interface with generated model code usually includes a customized main module (based on a main program provided by Real-Time Workshop Embedded Coder), and may also include interrupt handlers, device driver blocks and other S-functions, and other supervisory or supporting code.

You should establish a working directory for your own code modules. Your working directory should be on the MATLAB path. Minimally, you must also modify the ERT template makefile and system target file so that the build process can find your source and object files. More extensive modifications to the ERT target files are needed if you want to generate code for a particular microprocessor or development board, and to deploy the code on target hardware with a cross-development system.

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

Generating the Main Program Module

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



Code templates
Source file (*.c) template: ert_code_template.cgt
Header file (*.h) template: ert_code_template.cgt

Data templates
Source file (*.c) template: ert_code_template.cgt
Header file (*.h) template: ert_code_template.cgt

Custom templates
File customization template: example_file_process.tlc
[✓] Generate an example main program
Target operating system: BareBoardExample

Options for Generating a Main Program

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

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

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

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

The operation of the main program and the scheduling algorithm employed depend primarily upon whether your model is single-rate or multirate, and also upon your model's solver mode (`SingleTasking` vs. `MultiTasking`). These are described in detail in "Program Execution Overview" on page 1-11.

If you turn the **Generate an example main program** option off, Real-Time Workshop Embedded Coder provides a static version of the file `ert_main.c` as a basis for your custom modifications (see "Static Main Program Module" on page 1-26).

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

You can use a custom file processing (CFP) template file to override the normal main program generation, and generate a main program module customized for your target environment. To learn how to do this, see "Customizing Main Program Module Generation" on page 5-48.

Program Execution Overview

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 1-12 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 1-21 describes the operation of example programs designed for deployment under the VxWorks real-time operating system.
- “Model Entry Points” on page 1-24 describes the model entry-point functions that are generated for both stand-alone and VxWorks example programs.

Stand-Alone Program Execution

In this section...

Overview

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

To generate a stand-alone program:

1 In the **Custom templates** subpane of the **Real-Time Workshop/Templates** pane of the Configuration Parameters dialog box, select the **Generate an example main program** option (this option is on by default).

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

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

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

The execution driver, `rt_OneStep`, sequences calls to the `*model*_step` function(s). The operation of `rt_OneStep` differs depending on whether the generating model is single-rate or multirate. In a single-rate model, rt_OneStep simply calls the *model*_step function. In a multirate model, rt_OneStep prioritizes and schedules execution of blocks according to the rates at which they run.

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

- Whether the model executes in singletasking or multitasking mode.
- Whether or not reusable code is being generated.

These factors affect the scheduling algorithms used in generated code, and in some cases affect the API for the model entry point functions. The following sections discuss these variants.

Main Program

Overview of Operation

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 or .cpp program only partially implements this design. You must modify it according to your specifications.

Guidelines for Modifying the Main Program

This section describes the minimal modifications you should make in your production version of `ert_main.c` or `.cpp` to implement your harness program.

- After calling *`model`*`_initialize`:
  - Initialize target-specific data structures and hardware such as ADCs or DACs.
  - Install `rt_OneStep` as a timer ISR.
  - Initialize timer hardware.
  - Enable timer interrupts and start the timer.

Note `rtModel` is not in a valid state until *`model`*`_initialize` has been called. Servicing of timer interrupts should not begin until *`model`*`_initialize` has been called.

- Optionally, insert background task calls in the main loop.
- On termination of main loop (if applicable):
  - Disable timer interrupts.
  - Perform target-specific cleanup such as zeroing DACs.
  - Detect and handle errors. Note that even if your program is designed to run indefinitely, you may need to handle severe error conditions such as timer interrupt overruns.

    You can use the macros `rtmGetErrorStatus` and `rtmSetErrorStatus` to detect and signal errors.

rt_OneStep

Overview of Operation

The operation of `rt_OneStep` depends upon

- Whether your model is single-rate or multirate. In a single-rate model, the sample times of all blocks in the model, and the model's fixed step size, are the same. Any model in which the sample times and step size do not meet these conditions is termed multirate.

- Your model's solver mode (`SingleTasking` vs. `MultiTasking`)

Permitted Solver Modes for Real-Time Workshop Embedded Coder Targeted Models on page 1-15 summarizes the permitted solver modes for single-rate and multirate models. Note that for a single-rate model, only `SingleTasking` solver mode is allowed.

Permitted Solver Modes for Real-Time Workshop Embedded Coder Targeted Models

Mode	Single-Rate	Multirate
`SingleTasking`	Allowed	Allowed
`MultiTasking`	Disallowed	Allowed
`Auto`	Allowed (defaults to `SingleTasking`)	Allowed (defaults to `MultiTasking`)

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

Single-Rate Singletasking Operation

The only valid solver mode for a single-rate model is `SingleTasking`. Such models run in "single-rate" operation.

The following pseudocode shows the design of `rt_OneStep` in a single-rate program.

```
rt_OneStep()
{
  Check for interrupt overflow or other error
  Enable "rt_OneStep" (timer) interrupt
  Model_Step()  -- Time step combines output,logging,update
}
```

For the single-rate case, the generated *model*_step function is

```
void model_step(void)
```

Single-rate rt_OneStep is designed to execute *model*_step within a single clock period. To enforce this timing constraint, rt_OneStep maintains and checks a timer overrun flag. On entry, timer interrupts are disabled until the overrun flag and other error conditions have been checked. If the overrun flag is clear, rt_OneStep sets the flag, and proceeds with timer interrupts enabled.

The overrun flag is cleared only upon successful return from *model*_step. Therefore, if rt_OneStep is reinterrupted before completing *model*_step, the reinterruption is 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.

Multirate Multitasking Operation

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

The following pseudocode shows the design of rt_OneStep in a multirate multitasking program.

```
rt_OneStep()
{
  Check for base-rate interrupt overrun
  Enable "rt_OneStep" interrupt
  Determine which rates need to run this time step

  Model_Step0()        -- run base-rate time step code

  For N=1:NumTasks-1   -- iterate over sub-rate tasks
```

```
      If (sub-rate task N is scheduled)
      Check for sub-rate interrupt overrun
        Model_StepN()    -- run sub-rate time step code
      EndIf
    EndFor
  }
```

Task Identifiers. The execution of blocks having different sample rates is broken into tasks. Each block that executes at a given sample rate is assigned a *task identifier* (`tid`), which associates it with a task that executes at that rate. Where there are `NumTasks` tasks in the system, the range of task identifiers is 0..`NumTasks-1`.

Prioritization of Base-Rate and Sub-Rate Tasks. Tasks are prioritized, in descending order, by rate. The *base-rate* task is the task that runs at the fastest rate in the system (the hardware clock rate). The base-rate task has highest priority (`tid 0`). The next fastest task (`tid 1`) has the next highest priority, and so on down to the slowest, lowest priority task (`tid NumTasks-1`).

The slower tasks, running at submultiples of the base rate, are called *sub-rate* tasks.

Rate Grouping and Rate-Specific model_step Functions. In a single-rate model, all block output computations are performed within a single function, `model_step`. For multirate, multitasking models, Real-Time Workshop Embedded Coder uses a different strategy (whenever possible). This strategy is called *rate grouping*. Rate grouping generates separate `model_step` functions for the base rate task and each sub-rate task in the model. The function naming convention for these functions is

```
model_stepN
```

where `N` is a task identifier. For example, for a model named `my_model` that has three rates, the following functions are generated:

```
void my_model_step0 (void);
void my_model_step1 (void);
void my_model_step2 (void);
```

Each *model*_step*N* function executes all blocks sharing tid *N*; in other words, all block code that executes within task *N* is grouped into the associated *model*_step*N* function.

Scheduling model_stepN Execution. On each clock tick, rt_OneStep and *model*_step0 maintain scheduling counters and *event flags* for each sub-rate task. The counters are implemented in the Timing.TaskCounters.TID*n* fields of rtModel. The event flags are implemented as arrays, indexed on tid.

The scheduling counters are maintained by the rate_monotonic_scheduler function, which is called by *model*_step0 (that is, in the base-rate task). The function updates flags—an active task flag for each subrate and rate transition flags for tasks that exchange data—and assumes the use of a rate monotonic scheduler. The scheduling counters are, in effect, clock rate dividers that count up the sample period associated with each sub-rate task.

The event flags indicate whether or not a given task is scheduled for execution. rt_OneStep maintains the event flags with the *model*_SetEventsForThisBaseStep function. When a counter indicates that a task's sample period has elapsed, *model*_SetEventsForThisBaseStep sets the event flag for that task.

On each invocation, rt_OneStep updates its scheduling data structures and steps the base-rate task (rt_OneStep always calls *model*_step0 because the base-rate task must execute on every clock step). Then, rt_OneStep iterates over the scheduling flags in tid order, unconditionally calling *model*_step*N* for any task whose flag is set. This ensures that tasks are executed in order of priority.

Preemption. Note that the design of rt_OneStep assumes that interrupts are disabled before rt_OneStep is called. rt_OneStep should be noninterruptible until the base-rate interrupt overflow flag has been checked (see pseudocode above).

The event flag array and loop variables used by rt_OneStep are stored as local (stack) variables. This ensures that rt_OneStep is reentrant. If rt_OneStep is reinterrupted, higher priority tasks preempt lower priority tasks. Upon return from interrupt, lower priority tasks resume in the previously scheduled order.

Overrun Detection. Multirate `rt_OneStep` also maintains an array of timer overrun flags. `rt_OneStep` detects timer overrun, per task, by the same logic as single-rate `rt_OneStep`.

Note If you have developed multirate S-functions, or if you use a customized static main program module, see “Rate Grouping Compliance and Compatibility Issues” on page 1-32 for information about how to adapt your code for rate grouping compatibility. This adaptation lets your multirate, multitasking models generate more efficient code.

Multirate Singletasking Operation

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

In a multirate singletasking program, blocks execute at different rates, but under the same task identifier. The operation of `rt_OneStep`, in this case, is a simplified version of multirate multitasking operation. Rate grouping is not used. The only task is the base-rate task. Therefore, only one *`model`*`_step` function is generated:

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

On each clock tick, `rt_OneStep` checks the overrun flag and calls *`model`*`_step`, passing in `tid 0`. The scheduling function for a multirate singletasking program is `rate_scheduler` (rather than `rate_monotonic_scheduler`). The scheduler maintains scheduling counters on each clock tick. There is one counter for each sample rate in the model. The counters are implemented in an array (indexed on tid) within the `Timing` structure within `rtModel`.

The counters are, in effect, clock rate dividers that count up the sample period associated with each sub-rate task. When a counter indicates that a sample period for a given rate has elapsed, `rate_scheduler` clears the counter. This condition indicates that all blocks running at that rate should execute on the next call to *`model`*`_step`, which is responsible for checking the counters.

Guidelines for Modifying rt_OneStep

`rt_OneStep` does not require extensive modification. The only required modification is to re-enable interrupts after the overrun flag(s) and error conditions have been checked. If applicable, you should also

- Save and restore your FPU context on entry and exit to `rt_OneStep`.
- Set model inputs associated with the base rate before calling *`model`*`_step0`.
- Get model outputs associated with the base rate after calling *`model`*`_step0`.
- In a multirate, multitasking model, set model inputs associated with sub-rates before calling *`model`*`_step`*`N`* in the sub-rate loop.
- In a multirate, multitasking model, get model outputs associated with sub-rates after calling *`model`*`_step`*`N`* in the sub-rate loop.

Comments in `rt_OneStep` indicate the appropriate place to add your code.

In multirate `rt_OneStep`, you can improve performance by unrolling `for` and `while` loops.

In addition, you may choose to modify the overrun behavior to continue execution after error recovery is complete.

You should not modify the way in which the counters, event flags, or other timing data structures are set in `rt_OneStep`, or in functions called from `rt_OneStep`. The `rt_OneStep` timing data structures (including `rtModel`) and logic are critical to correct operation of any Real-Time Workshop Embedded Coder program.

VxWorks Example Main Program Execution

In this section...
"Overview" on page 1-21
"Task Management" on page 1-21

Overview

The Real-Time Workshop Embedded Coder VxWorks example main program is provided as a template for the deployment of generated code in a real-time operating system (RTOS) environment. You should read the preceding sections of this chapter as a prerequisite to working with the VxWorks example main program. An understanding of the Real-Time Workshop Embedded Coder scheduling and tasking concepts and algorithms, described in "Stand-Alone Program Execution" on page 1-12, 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 **Custom templates** subpane of the **Real-Time Workshop/Templates** pane of the Configuration Parameters dialog box, select the **Generate an example main program** option (this option is on by default).

2. When **Generate an example main program** is selected, the **Target operating system** pop-up menu is enabled. Select `VxWorksExample` from this menu.

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

Task Management

In a VxWorks example program, the main program and the base rate and sub-rate tasks (if any) run as prioritized tasks under VxWorks. The logic of a

VxWorks example program parallels that of a stand-alone program; the main difference lies in the fact that base rate and sub-rate tasks are activated by clock semaphores managed by the operating system, rather than directly by timer interrupts.

Your application code must spawn `*model*_main()` as an independent VxWorks task. The task priority you specify is passed in to `*model*_main()`.

As with a stand-alone program, the VxWorks example program architecture is tailored to the number of rates in the model and to the solver mode (see Permitted Solver Modes for Real-Time Workshop Embedded Coder Targeted Models on page 1-15). 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`.

Multirate Multitasking Operation

In a multirate, multitasking model, `*model*_main()` spawns a base rate task and sub-rate tasks. Task priorities are assigned by rate.

As in a stand-alone program, rate grouping code is used (where possible) for multirate, multitasking models. The base rate task calls `*model*_step0`, while the sub-rate tasks call `*model*_step*N*`. The base rate task calls a function that updates flags—an active task flag for each subrate and rate transition flags for tasks that exchange data. This function assumes the use of a rate-monotonic scheduler.

Multirate Singletasking Operation

In a multirate, singletasking model, `*model*_main()` spawns only a base rate task, `tBaseRate`. All rates run under this task. The base rate task is activated by a clock semaphore provided by VxWorks, rather than by a timer interrupt. On each activation, `tBaseRate` calls *`model`*`_step`.

`model``_step` in turn calls the `rate_scheduler` utility, which maintains the scheduling counters that determine which rates should execute. *`model`*`_step` is responsible for checking the counters.

Model Entry Points

The following functions represent entry points in the generated code for a Simulink model.

Function	Description
`model_initialize`	Initialization entry point in generated code for Simulink model
`model_SetEventsForThisBaseStep`	Set event flags for multirate, multitasking operation before calling *`model`*`_step` for Simulink model
`model_step`	Step routine entry point in generated code for Simulink model
`model_terminate`	Termination entry point in generated code for Simulink model

Note that the calling interface generated for each of these functions differs significantly depending on how you set the **Generate reusable code** option (see "Configuring Model Interfaces" on page 2-22).

By default, **Generate reusable code** is off, and the model entry point functions access model data with statically allocated global data structures.

When **Generate reusable code** is on, model data structures are passed in (by reference) as arguments to the model entry point functions. For efficiency, only those data structures that are actually used in the model are passed in. Therefore when **Generate reusable code** is on, the argument lists generated for the entry point functions vary according to the requirements of the model.

The entry points are exported with *`model`*`.h`. To call the entry-point functions from your hand-written code, add an `#include` *`model`*`.h` directive to your code. If **Generate reusable code** is on, you must examine the generated code to determine the calling interface required for these functions.

For more information, see the reference pages for the listed functions.

Note The function reference pages document the default (**Generate reusable code** off) calling interface generated for these functions.

Static Main Program Module

In this section...

Overview

In most cases, the easiest strategy for deploying your generated code is to use the **Generate an example main program option** to generate the `ert_main.c` or `.cpp` module (see "Generating the Main Program Module" on page 1-9).

However, if you turn the **Generate an example main program** option off, you can use the module *`matlabroot`*`/rtw/c/ert/ert_main.c` as a template example for developing your embedded applications. The module is not part of the generated code; it is provided as a basis for your custom modifications, and for use in simulation. If your existing applications, developed prior to this release, depend upon a static `ert_main.c`, you may need to continue using this module.

When developing applications using a static `ert_main.c`, you should copy this module to your working directory and rename it to *`model`*`_ert_main.c` before making modifications. Also, you must modify the template makefile such that the build process creates *`model`*`_ert_main.obj` (on Unix, *`model`*`_ert_main.o`) in the build directory.

The static `ert_main.c` contains

- `rt_OneStep`, a timer interrupt service routine (ISR). `rt_OneStep` calls *`model`*`_step` to execute processing for one clock period of the model.
- A skeletal `main` function. As provided, `main` is useful in simulation only. You must modify `main` for real-time interrupt-driven execution.

For single-rate models, the operation of `rt_OneStep` and the main function are essentially the same in the static version of `ert_main.c` as they are in the autogenerated version described in “Stand-Alone Program Execution” on page 1-12. For multirate, multitasking models, however, the static and generated code is slightly different. The next section describes this case.

Rate Grouping and the Static Main Program

Targets based on the ERT target sometimes use a static `ert_main` module and disallow use of the **Generate an example main program** option. This may be necessary because target-specific modifications have been added to the static `ert_main.c`, and these modifications would not be preserved if the main program were regenerated.

Your `ert_main` module may or may not use rate grouping compatible *`model`*`_step`*`N`* functions. If your `ert_main` module is based on the static `ert_main.c` module, it does not use rate-specific *`model`*`_step`*`N`* function calls. The static `ert_main.c` module uses the old-style *`model`*`_step` function, passing in a task identifier:

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

By default, when the **Generate an example main program** option is off, the ERT target generates a *`model`*`_step` “wrapper” for multirate, multitasking models. The purpose of the wrapper is to interface the rate-specific *`model`*`_step`*`N`* functions to the old-style call. The wrapper code dispatches to the appropriate *`model`*`_step`*`N`* call with a `switch` statement, as in the following example:

```
void mymodel_step(int_T tid) /* Sample time:  */
{

  switch(tid) {
   case 0 :
    mymodel_step0();
    break;
   case 1 :
    mymodel_step1();
    break;
   case 2 :
    mymodel_step2();
    break;
   default :
    break;
  }
}
```

The following pseudocode shows how `rt_OneStep` calls *`model`*`_step` from the static main program in a multirate, multitasking model.

```
rt_OneStep()
{
  Check for base-rate interrupt overflow
  Enable "rt_OneStep" interrupt
  Determine which rates need to run this time step

  ModelStep(tid=0)     --base-rate time step

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

You can use the TLC variable `RateBasedStepFcn` to specify that only the rate-based step functions are generated, without the wrapper function. If your target calls the rate grouping compatible *`model`*`_step`*`N`* function directly, set `RateBasedStepFcn` to `1`. In this case, the wrapper function is not generated.

You should set `RateBasedStepFcn` prior to the `%include "codegenentry.tlc"` statement in your system target file. Alternatively, you can set `RateBasedStepFcn` in your `target_settings.tlc` file.

Modifying the Static Main Program

As in the generated `ert_main.c`, a few modifications to the main loop and `rt_OneStep` are necessary. See "Guidelines for Modifying the Main Program" on page 1-14 and "Guidelines for Modifying rt_OneStep" on page 1-20.

Also, you should replace the `rt_OneStep` call in the main loop with a background task call or null statement.

Other modifications you may need to make are

- If your model has multiple rates, the generated code does not operate correctly unless:
  - The multirate scheduling code is removed. The relevant code is tagged with the keyword `REMOVE` in comments (see also the Version 3.0 comments in `ert_main.c`).
  - Use the `MODEL_SETEVENTS` macro (defined in `ert_main.c`) to set the event flags instead of accessing the flags directly. The relevant code is tagged with the keyword `REPLACE` in comments.
- Remove old `#include ertformat.h` directives. `ertformat.h` will be obsoleted in a future release. The following macros, formerly defined in `ertformat.h`, are now defined within `ert_main.c`:

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

  See also the comments in `ertformat.h`.
- If applicable, follow comments in the code regarding where to add code for reading/writing model I/O and saving/restoring FPU context.
- When the **Generate an example main program** option is off, Real-Time Workshop Embedded Coder generates the file `autobuild.h` to provide an interface between the main module and generated model code. If you create your own static main program module, you would normally include `autobuild.h`.

  Alternatively, you can suppress generation of `autobuild.h`, and include *`model`*`.h` directly in your main module. To suppress generation of `autobuild.h`, use the following statement in your system target file:

  ```
  %assign AutoBuildProcedure = 0
  ```

- If you have cleared the **Terminate function required** option, remove or comment out the following in your production version of `ert_main.c`:
  - The `#if TERMFCN...` compile-time error check
  - The call to `MODEL_TERMINATE`
- If you do *not* want to combine output and update functions, clear the **Single output/update function** option and make the following changes in your production version of `ert_main.c`:
  - Replace calls to `MODEL_STEP` with calls to `MODEL_OUTPUT` and `MODEL_UPDATE`.
  - Remove the `#if ONESTEPFCN...` error check.
- The static `ert_main.c` module does not support the **Generate Reusable Code** option. Use this option only if you are generating a main program. The following error check raises a compile-time error if **Generate Reusable Code** is used illegally.

  ```
  #if MULTI_INSTANCE_CODE==1
  ```

- The static `ert_main.c` module does not support the **External mode** option. Use this option only if you are generating a main program. The following error check raises a compile-time error if **External mode** is used illegally.

  ```
  #ifdef EXT_MODE
  ```

Rate Grouping Compliance and Compatibility Issues

In this section...

Main Program Compatibility

When the **Generate an example main program** option is off, Real-Time Workshop Embedded Coder generates slightly different rate grouping code, for compatibility with the older static `ert_main.c` module. See "Rate Grouping and the Static Main Program" on page 1-28 for details.

Making Your S-Functions Rate Grouping Compliant

All built-in Simulink blocks, as well as all blocks in Signal Processing Blockset, are compliant with the requirements for generating rate grouping code. However, user-written multirate inlined S-functions may not be rate grouping compliant. Non-compliant blocks generate less efficient code, but are otherwise compatible with rate grouping. To take full advantage of the efficiency of rate grouping, your multirate inlined S-functions must be upgraded to be fully rate grouping compliant. You should upgrade your TLC S-function implementations, as described in this section.

Use of non-compliant multirate blocks to generate rate-grouping code generates dead code. This can cause two problems:

- Reduced code efficiency.
- Warning messages issued at compile time. Such warnings are caused when dead code references temporary variables before initialization. Since the dead code never runs, this problem does not affect the run-time behavior of the generated code.

To make your S-functions rate grouping compliant, you can use the following TLC functions to generate `ModelOutputs` and `ModelUpdate` code, respectively:

```
OutputsForTID(block, system, tid)
UpdateForTID(block, system, tid)
```

The code listings below illustrate generation of output computations without rate grouping (Listing 1) and with rate grouping (Listing 2). Note the following:

- The `tid` argument is a task identifier (`0..NumTasks-1`).
- Only code guarded by the `tid` passed in to `OutputsForTID` is generated. The `if (%<LibIsSFcnSampleHit(portName)>)` test is not used in `OutputsForTID`.
- When generating rate grouping code, `OutputsForTID` and/or `UpdateForTID` is called during code generation. When generating non-rate-grouping code, `Outputs` and/or `Update` is called.
- In rate grouping compliant code, the top-level `Outputs` and/or `Update` functions call `OutputsForTID` and/or `UpdateForTID` functions for each rate (`tid`) involved in the block. The code returned by `OutputsForTID` and/or `UpdateForTID` must be guarded by the corresponding `tid` guard:

  ```
  if (%<LibIsSFcnSampleHit(portName)>)
  ```

  as in Listing 2.

Listing 1: Outputs Code Generation Without Rate Grouping

```
%% multirate_blk.tlc

%implements "multirate_blk" "C"


%% Function: mdlOutputs =====================================================
%% Abstract:
%%
%%  Compute the two outputs (input signal decimated by the
%%  specified parameter). The decimation is handled by sample times.
%%  The decimation is only performed if the block is enabled.
%%  Each ports has a different rate.
%%
%%  Note, the usage of the enable should really be protected such that
%%  Neach task has its own enable state. In this example, the enable
%% occurs immediately which may or may not be the expected behavior.
```

```
%%
  %function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign enable = LibBlockInputSignal(0, "", "", 0)
  {
    int_T *enabled = &%<LibBlockIWork(0, "", "", 0)>;

    %if LibGetSFcnTIDType("InputPortIdx0") == "continuous"
      %% Only check the enable signal on a major time step.
      if (%<LibIsMajorTimeStep()> && ...
                          %<LibIsSFcnSampleHit("InputPortIdx0")>) {
        *enabled = (%<enable> > 0.0);
      }
    %else
      if (%<LibIsSFcnSampleHit("InputPortIdx0")>) {
        *enabled = (%<enable> > 0.0);
      }
    %endif

    if (*enabled) {
      %assign signal = LibBlockInputSignal(1, "", "", 0)
      if (%<LibIsSFcnSampleHit("OutputPortIdx0")>) {
        %assign y = LibBlockOutputSignal(0, "", "", 0)
        %<y> = %<signal>;
      }
      if (%<LibIsSFcnSampleHit("OutputPortIdx1")>) {
        %assign y = LibBlockOutputSignal(1, "", "", 0)
        %<y> = %<signal>;
      }
    }
  }

  %endfunction
%% [EOF] sfun_multirate.tlc
```

Listing 2: Outputs Code Generation With Rate Grouping

```
%% example_multirateblk.tlc

%implements "example_multirateblk" "C"
```

```
%% Function: mdlOutputs =====================================================
%% Abstract:
%%
%% Compute the two outputs (the input signal decimated by the
%% specified parameter). The decimation is handled by sample times.
%% The decimation is only performed if the block is enabled.
%% All ports have different sample rate.
%%
%% Note: the usage of the enable should really be protected such that
%% each task has its own enable state. In this example, the enable
%% occurs immediately which may or may not be the expected behavior.
%%
%function Outputs(block, system) Output



%assign portIdxName = ["InputPortIdx0","OutputPortIdx0","OutputPortIdx1"]
%assign portTID     = [%<LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx0")>, ...
                      %<LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx0")>, ...
                      %<LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx1")>]
%foreach i = 3
  %assign portName = portIdxName[i]
  %assign tid      = portTID[i]
  if (%<LibIsSFcnSampleHit(portName)>) {
                     %<OutputsForTID(block,system,tid)>
  }
%endforeach

%endfunction

%function OutputsForTID(block, system, tid) Output
/* %<Type> Block: %<Name> */
%assign enable = LibBlockInputSignal(0, "", "", 0)
%assign enabled = LibBlockIWork(0, "", "", 0)
%assign signal = LibBlockInputSignal(1, "", "", 0)

%switch(tid)
  %case LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx0")
```

```
                              %if LibGetSFcnTIDType("InputPortIdx0") == "continuous"
                                %% Only check the enable signal on a major time step.
                                if (%<LibIsMajorTimeStep()>) {
                                  %<enabled> = (%<enable> > 0.0);
                                }
                              %else
                                %<enabled> = (%<enable> > 0.0);
                              %endif
                              %break
    %case LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx0")
                              if (%<enabled>) {
                                %assign y = LibBlockOutputSignal(0, "", "", 0)
                                %<y> = %<signal>;
                              }
                              %break
    %case LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx1")
                              if (%<enabled>) {
                                %assign y = LibBlockOutputSignal(1, "", "", 0)
                                %<y> = %<signal>;
                              }
                              %break
    %default
                              %% error it out
  %endswitch

  %endfunction

%% [EOF] sfun_multirate.tlc
```

Code Generation Options and Optimizations

Accessing the ERT Target Options

In this section...
"Introduction" on page 2-3
"Viewing ERT Target Options in the Configuration Parameters Dialog Box or Model Explorer" on page 2-4

Introduction

This chapter explains how to use the Embedded Real-Time (ERT) target code generation options to configure models for production code generation. The discussion also includes other options that are not specific to the ERT target, but which affect ERT code generation.

Every model contains one or more named configuration sets that specify model parameters such as solver options, code generation options, and other choices. A model can contain multiple configuration sets, but only one configuration set is active at any time. A configuration set includes code generation options that affect Real-Time Workshop in general, and options that are specific to a given target, such as the ERT target.

Configuration sets can be particularly useful in embedded systems development. By defining multiple configuration sets in a model, you can easily retarget code generation from that model. For example, one configuration set might specify the default ERT target with external mode support enabled for rapid prototyping, while another configuration set might specify the Target for Freescale™ MPC5xx to generate production code for deployment of the application. Activation of either configuration set fully reconfigures the model for the appropriate type of code generation.

Before you work with the ERT target options, you should become familiar with

- Configuration sets and how to view and edit them in the Configuration Parameters dialog box. The Using Simulink document contains detailed information on these topics.
- Real-Time Workshop code generation options and the use of the System Target File Browser. The Real-Time Workshop documentation contains detailed information on these topics.

For descriptions of the Embedded Real-Time (ERT) target code generation options, see “Configuration Parameters” in the Real-Time Workshop Embedded Coder reference documentation.

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

The Configuration Parameters dialog box and Model Explorer provide the quickest routes to a model’s active configuration set. Illustrations throughout this chapter and “Configuration Parameters” in the Real-Time Workshop Embedded Coder reference documentation show the Configuration Parameters dialog box view of model parameters (unless otherwise noted).

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

In this section...

Generating Code for Continuous Time Blocks

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

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

```
showblockdatatypetable
```

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

Generating Code that Supports Continuous Solvers

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

Note Custom targets must be modified to support continuous time. The required modifications are described in the Developing Embedded Targets document.

Generating Code that Honors a Stop Time

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

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

Otherwise, the executable runs indefinitely.

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

Mapping Application Requirements to Configuration Options

The first step to applying Real-Time Workshop Embedded Coder to the application development process is to consider how your application requirements, particularly with respect to traceability, efficiency, and safety, map to code generation options in a model configuration set.

Parameters that you set in the **Solver**, **Data Import/Export**, **Diagnostics**, and **Real-Time Workshop** panes of the Configuration Parameters dialog box affect the behavior of a model in simulation and the code generated for the model.

Consider questions such as the following:

- What settings might help you debug your application?
- What is the highest priority for your application — debugging, traceability, efficiency, extra safety precaution, or some other criteria?
- What is the second highest priority?
- Can the priority at the start of the project differ from the priority required for the end result? What tradeoffs can be made?

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

Configuring a Model

In this section...

Selecting an ERT Target

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

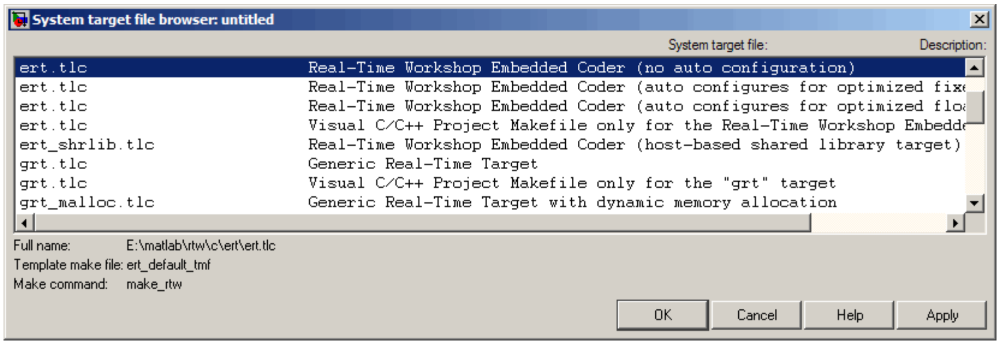
To make it easier for you to generate code that is optimized for your target hardware, Real-Time Workshop Embedded Coder provides three variants of the ERT target that

- Automatically configure parameters that are optimized for fixed-point code generation
- Automatically configure parameters that are optimized for floating-point code generation

- Applies default parameter settings

The discussion throughout this chapter assumes use of the default ERT target.

These targets are based on a common system target file, `ert.tlc`. They are displayed in the System Target File Browser as shown in the figure below.



```
ert.tlc            Real-Time Workshop Embedded Coder (no auto configuration)
ert.tlc            Real-Time Workshop Embedded Coder (auto configures for optimized fixe
ert.tlc            Real-Time Workshop Embedded Coder (auto configures for optimized flo
ert.tlc            Visual C/C++ Project Makefile only for the Real-Time Workshop Embedde
ert_shrlib.tlc     Real-Time Workshop Embedded Coder (host-based shared library target)
grt.tlc            Generic Real-Time Target
grt.tlc            Visual C/C++ Project Makefile only for the "grt" target
grt_malloc.tlc     Generic Real-Time Target with dynamic memory allocation
```

Full name: E:\matlab\rtw\c\ert\ert.tlc
Template make file: ert_default_tmf
Make command: make_rtw

The optimized ERT target variants are discussed in detail in "Generating Efficient Code with Optimized ERT Targets" on page 5-26.

You can implement a custom auto-configuring target, using the same mechanism used by the optimized ERT target variants. "Auto-Configuring Models for Code Generation" on page 5-22 discusses the auto-configuration mechanism and utilities used by the optimized ERT target variants.

You can use the `ert_shrlib.tlc` target to generate a host-based shared library from your Simulink model. Selecting this target allows you to generate a shared library version of your model code that is appropriate for your host platform, either a Windows dynamic link library (`.dll`) file or a UNIX shared object (`.so`) file. This feature can be used to package your source code securely for easy distribution and shared use. For more information, see "Creating and Using Host-Based Shared Libraries" on page 2-104.

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

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

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

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

Customizing Comments in Generated Code

You can customize the comments in the generated code for ERT targets by setting or clearing several parameters on the **Real-Time Workshop > Comments** pane. These options let you enable or suppress generation of descriptive information in comments for blocks and other objects in the model.

To...	Select...
Include the text specified in the **Description** field of a block's Block Properties dialog box as comments in the code generated for each block	**Simulink block descriptions**.
Add a comment that includes the blockname at the start of the code for each block	**Simulink block descriptions**
Include the text specified in the **Description** field of a Simulink data object (such as a signal, parameter, data type, or bus) in the Simulink Model Explorer as comments in the code generated for each object	**Simulink data object descriptions**.
Include comments just above signals and parameter identifiers in the generated code as specified in an M-code or TLC function.	**Custom comments (MPT objects only)**.
Include the text specified in the **Description** field of the Properties dialog box for a Stateflow object as comments just above the code generated for each object	**Stateflow object descriptions** .
Include requirements assigned to Simulink blocks in the generated code comments (for more information, see "Including Requirements with Generated Code" in the Simulink Verification and Validation documentation)	**Requirements in block comments**.

When you select **Simulink block descriptions**,

- The description text for blocks and Stateflow objects and block names generated as comments can include international (non-US-ASCII) characters. (For details on international character support, see "Support for International (Non-US-ASCII) Characters” in the Real-Time Workshop documentation.)

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

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

Customizing Generated Identifiers

Several parameters are available for customizing generated symbols.

To...	Specify...
Define a macro string that specifies whether, and in what order, certain substrings are included within generated identifiers for global variables, global types, field names of global types, subsystem methods, local temporary variables, local block output variables, and constant macros	The macro string with the **Identifier format control** parameter (for details on how to specify formats, see "Specifying Identifier Formats" on page 2-14 and for limitations, see "Identifier Format Control Parameters Limitations" on page 2-20).
Specify the minimum number of characters the code generator uses for mangled symbols	Specify an integer value for the **Minimum mangle length** (for details, see "Name Mangling" on page 2-17).
Specify the maximum number of characters the code generator can use for function, `typedef`, and variable names (default 31)	Specify an integer value for the **Maximum identifier length**. If you expect your model to generate lengthy identifiers (due to use of long signal or parameter names, for example), or you find that identifiers are being mangled more than expected, you should increase the value of this parameter.
Control whether scalar inlined parameter values are expressed in generated code as literal values or macros	The value `Literals` or `Macros` for the **Generate scalar inlined parameters as** parameter . • `Literals`: Parameters are expressed as numeric constants and takes effect if **Inline parameters** is selected. • `Macros`: Parameters are expressed as variables (with `#define` macros). This setting makes code more readable.

For more information, see "Real-Time Workshop Pane: Symbols" in the Real-Time Workshop reference documentation.

Configuring Symbols

Specifying Simulink Data Object Naming Rules

To Define Rules that Change the Names of a Model's...	Specify a Naming Rule with the ...
Signals	**Signal naming** parameter
Parameters	**Parameter naming** parameter
Parameters that have a storage class of `Define`	**#define naming** parameter

For more information on these parameters, see "Specifying Simulink Data Object Naming Rules" in the Real-Time Workshop Embedded Coder Module Packaging Features document.

Specifying Identifier Formats

The **Identifier format control** parameters let you customize generated identifiers by entering a macro string that specifies whether, and in what order, certain substrings are included within generated identifiers. For example, you can specify that the root model name be inserted into each identifier.

The macro string can include

- Tokens of the form `$X`, where `X` is a single character. Valid tokens are listed in Identifier Format Tokens on page 2-15. You can use or omit tokens as you want, with the exception of the `$M` token, which is required (see "Name Mangling" on page 2-17) and subject to the use and ordering restrictions noted in Identifier Format Control Parameter Values on page 2-16.
- Any valid C or C++ language identifier characters (`a-z, A-Z, _ , 0-9`).

The build process generates each identifier by expanding tokens (in the order listed in Identifier Format Tokens on page 2-15) and inserting the resultant strings into the identifier. Character strings between tokens are simply inserted directly into the identifier. Contiguous token expansions are separated by the underscore (_) character.

Identifier Format Tokens

Token	Description
`$M`	Insert name mangling string if required to avoid naming collisions (see "Name Mangling" on page 2-17). **Note:** This token is required.
`$F`	Insert method name (for example, `_Update` for update method). This token is available only for subsystem methods.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Note that when using model referencing, this token is required in addition to `$M` (see "Model Referencing Considerations" on page 2-20). **Note:** This token replaces the **Prefix model name to global identifiers** option used in previous releases.

Identifier Format Tokens (Continued)

Token	Description
`$H`	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the string `root_`. For blocks at the subsystem level, the tag is of the form `sN_`, where `N` is a unique system number assigned by Simulink. This token is available only for subsystem methods and field names of global types. **Note:** This token replaces the **Include System Hierarchy Number in Identifiers** option used in previous releases.
`$A`	Insert data type acronym (for example, `i32` for long integers) to signal and work vector identifiers. This token is available only for local block output variables and field names of global types. **Note:** This token replaces the **Include data type acronym in identifier** option used in previous releases.

Identifier Format Control Parameter Values on page 2-16 lists the default macro string, the supported tokens, and the applicable restrictions for each **Identifier format control** parameter.

Identifier Format Control Parameter Values

Parameter	Default Value	Supported Tokens	Restrictions
Global variables	`$R$N$M`	`$R`, `$N`, `$M`	`$F`, `$H`, and `$A` are disallowed.
Global types	`$N$R$M`	`$N`, `$R`, `$M`	`$F`, `$H`, and `$A` are disallowed.
Field name of global types	`$N$M`	`$N`, `$M`, `$H`, `$A`	`$R` and `$F` are disallowed.
Subsystem methods	`$R$N$M$F`	`$R`, `$N`, `$M`, `$F`, `$H`	`$F` and `$H` are empty for Stateflow functions; `$A` is disallowed.

Identifier Format Control Parameter Values (Continued)

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

Non-ERT based targets (such as the GRT target) implicitly use a default `$R$N$M` specification. This specifies identifiers consisting of the root model name, followed by the name of the generating object (signal, parameter, state, and so on), followed by a name mangling string (see “Name Mangling” on page 2-17).

For limitations that apply to **Identifier format control** parameters, see “Identifier Format Control Parameters Limitations” on page 2-20.

Name Mangling

In identifier generation, a circumstance that would cause generation of two or more identical identifiers is called a *name collision*. Name collisions are never permissible. When a potential name collision exists, unique *name mangling* strings are generated and inserted into each of the potentially conflicting identifiers. Each name mangling string is guaranteed to be unique for each generated identifier.

The position of the `$M` token in the **Identifier format control** parameter specification determines the position of the name mangling string in the generated identifiers. For example, if the specification `$R$N$M` is used, the name mangling string is appended (if required) to the end of the identifier.

The **Minimum mangle length** parameter specifies the minimum number of characters used when a name mangling string is generated. The default is 1 character. As described below, the actual length of the generated string may be longer than this minimum.

Traceability

An important aspect of model based design is the ability to generate identifiers that can easily be traced back to the corresponding entities within the model. To ensure traceability, it is important to make sure that incremental revisions to a model have minimal impact on the identifier names that appear in generated code. There are two ways of achieving this in Real-Time Workshop Embedded Coder:

1. Choose unique names for objects in Simulink (blocks, signals, states, and so on) as much as possible.

2. Make use of name mangling when conflicts cannot be avoided.

When conflicts cannot be avoided (as may be the case in models that use libraries or model reference), name mangling ensures traceability. The position of the name mangling string is specified by the placement of the `$M` token in the **Identifier format control** parameter specification. Mangle characters consist of lower case characters (`a-z`) and numerics (`0-9`), which are chosen with a checksum that is unique to each object. How Name Mangling Strings Are Computed on page 2-18 describes how this checksum is computed for different types of objects.

How Name Mangling Strings Are Computed

Object Type	Source of Mangling String
Block diagram	Name of block diagram
Simulink block	Full path name of block
Simulink parameter	Full name of parameter owner (that is, model or block) and parameter name

How Name Mangling Strings Are Computed (Continued)

Object Type	Source of Mangling String
Simulink signal	Signal name, full name of source block, and port number
Stateflow objects	Complete path to Stateflow block and Stateflow computed name (unique within chart)

The length of the name mangling string is specified by the **Minimum mangle length** parameter. The default value is `1`, but this automatically increases during code generation as a function of the number of collisions.

To minimize disturbance to the generated code during development, specify a larger **Minimum mangle length**. A **Minimum mangle length** of `4` is a conservative and safe value. A value of `4` allows for over 1.5 million collisions for a particular identifier before the mangle length is increased.

Minimizing Name Mangling

Note that the length of generated identifiers is limited by the **Maximum identifier length** parameter. When a name collision exists, the `$M` token is always expanded to the minimum number of characters required to avoid the collision. Other tokens and character strings are expanded in the order listed in Identifier Format Tokens on page 2-15. If the **Maximum identifier length** is not large enough to accommodate full expansions of the other tokens, partial expansions are used. To avoid this outcome, it is good practice to

- Avoid name collisions in general. One way to do this is to avoid using default block names (for example, `Gain1`, `Gain2...`) when there are many blocks of the same type in the model.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate.

  Set the **Minimum mangle length** parameter to reserve at least three characters for the name mangling string. The length of the name mangling string increases as the number of name collisions increases.

  Note that an existing name mangling string increases (or decreases) in length if changes to model create more (or fewer) collisions. If the length of the name mangling string increases, additional characters are appended to the existing string. For example, `'xyz'` might change to `'xyzQ'`. In the inverse case (fewer collisions) `'xyz'` would change to `'xy'`.

Model Referencing Considerations

Within a model that uses model referencing, there can be no collisions between the names of the constituent models. When generating code from a model that uses model referencing:

- The `$R` token must be included in the **Identifier format control** parameter specifications (in addition to the `$M` token).
- The **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens. A code generation error occurs if **Maximum identifier length** is not large enough.

When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the identifier from the referenced model is preserved. Name mangling is performed on the identifier from the higher-level model.

Exceptions to Identifier Formatting Conventions

There are some exceptions to the identifier formatting conventions described above:

- Type name generation: The above name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. If the `$R` token is included in the **Identifier format control** parameter specification, the model name is included in the `typedef`. The **Maximum identifier length** parameter is not respected when generating type definitions.
- Non-`Auto` storage classes: The **Identifier format control** parameter specification does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Identifier Format Control Parameters Limitations

The following limitations apply to the **Identifier format control** parameters:

- The following auto-generated identifiers currently do not fully comply with the setting of the **Maximum identifier length** parameter on the **Real-Time Workshop/Symbols** pane of the Configuration Parameters dialog box.
  - Model methods
    - The applicable format string is `$R$F`, and the longest `$F` is `_derivatives`, which is 12 characters long. The model name can be up to 19 characters without exceeding the default **Maximum identifier length** of 31.
  - Local functions generated by S-functions or by add-on products such as Signal Processing Blockset that rely on S-functions
  - Local variables generated by S-functions or by add-on products such as Signal Processing Blockset that rely on S-functions
  - `DWork` identifiers generated by S-functions in referenced models
  - Fixed-point shared utility macros or shared utility functions
  - Simulink `rtm` macros
    - Most are within the default **Maximum identifier length** of 31, but some exceed the limit. Examples are `RTMSpecAccsGetStopRequestedValStoredAsPtr`, `RTMSpecAccsGetErrorStatusPointer`, and `RTMSpecAccsGetErrorStatusPointerPointer`.
  - Define protection guard macros
    - Header file guards, such as `_RTW_HEADER_$(filename)_h_`, which can exceed the default **Maximum identifier length** of 31 given a filename such as `$R_private.h`.
    - Include file guards, such as `_$R_COMMON_INCLUDES_`.
    - Typedef guards, such as `_CSCI_$R_CHARTSTRUCT_`.
- In some situations, the following identifiers potentially can conflict with others.
  - Model methods
  - Local functions generated by S-functions or by add-on products such as Signal Processing Blockset that rely on S-functions

  - Local variables generated by S-functions or by add-on products such as Signal Processing Blockset that rely on S-functions
  - Fixed-point shared utility macros or shared utility functions
  - Include header guard macros

- The following external identifiers that are unknown to Simulink may potentially conflict with auto-generated identifiers.
  - Identifiers defined in custom code
  - Identifiers defined in custom header files
  - Identifiers introduced through a non-ANSI-C standard library
  - Identifiers defined by custom TLC code
- Identifiers generated for simulation targets may exceed the **Maximum identifier length**. Simulation targets include the model reference simulation target, the accelerated simulation target, the RSim target, and the S-function target.

Configuring Model Interfaces

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

- “Configuring Support for Numeric Data” on page 2-22
- “Configuring Support for Time Values” on page 2-23
- “Configuring Support for Non-Inlined S-Functions” on page 2-24
- “Configuring Model Function Generation and Argument Passing” on page 2-24
- “Configuring a Model for Code Reuse” on page 2-26

Configuring Support for Numeric Data

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

To Generate Code that Supports...	Do...
Integer data only	Deselect **Support floating-point numbers**. If any noninteger data or expressions are encountered during code generation, an error message reports the offending blocks and parameters.
Floating-point data	Select **Support floating-point numbers**.
Non-finite values (for example, `NaN`, `Inf`)	Select **Support floating-point numbers** and **Support non-finite numbers** .
Complex data	Select **Support complex numbers** .

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

Configuring Support for Time Values

Certain blocks require the value of absolute time (that is, the time from the start of program execution to the present time) , elapsed time (for example, the time elapsed between two trigger events), or continuous time. Depending on the blocks used, you might need to adjust the configuration settings for supported time values.

To...	Select...
Generate code that creates and maintains integer counters for blocks that use absolute or elapsed time values (default)	**Support absolute time**. For further information on the allocation and operation of absolute and elapsed timers, see the "Timing Services" chapter of the Real-Time Workshop documentation. If you do not select this parameter and the model includes block that use absolute or elapsed time values, the build process generates an error.
Generate code for blocks that rely on continuous time	**Support continuous time**. If you do not select this parameter and the model includes continuous-time blocks, the build process generates an error.

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

Configuring Support for Non-Inlined S-Functions

To generate code for non-inlined S-Functions in a model, select **Support non-inlined S-functions**. The generation of non-inlined S-functions requires floating-point and non-finite numbers. Thus, when you select **Support non-inlined S-functions**, the ERT target automatically selects **Support floating-point numbers** and **Support non-finite numbers**.

When you select **Support non-finite numbers**, the build process generates an error if the model includes a C-MEX S-function that does not have a corresponding TLC implementation (for inlining code generation).

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

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

Configuring Model Function Generation and Argument Passing

For ERT targets, you can configure how a model's functions are generated and how arguments are passed to the functions.

To...	Do...
Generate model function calls that are compatible with the main program module of the GRT target (`grt_main.c` or `.cpp`)	Select **GRT compatible call interface** and **MAT-file logging** . In addition, deselect **Suppress error status in real-time model data structure**. **GRT compatible call interface** provides a quick way to use ERT target features with a GRT-based custom target by generating wrapper function calls that interface to the ERT target's `Embedded-C` formatted code.
Reduce overhead and use more local variables by combining the output and update functions in a single *`model`*`_step` function	Select **Single output/update function** Errors or unexpected behavior can occur if a Model block is part of a cycle and "Single output/update function" is enabled (the default). See "Model Blocks and Direct Feedthrough" for details.

To...	Do...
Generate a *model*_terminate function for a model not designed to run indefinitely	Select **Terminate function required**. For more information, see the description of `model_terminate`.
Generate reusable, reentrant code from a model or subsystem	Select **Generate reusable code**. See “Configuring a Model for Code Reuse” on page 2-26 for details.
Statically allocate model data structures and access them directly in the model code	Deselect **Generate reusable code**. The generated code is not reusable or reentrant. See “Model Entry Points” on page 1-24 for information on the calling interface generated for model functions in this case.
Suppress the generation of an error status field in the real-time model data structure, `rtModel`, for example, if you do not need to log or monitor error messages	Select **Suppress error status in real-time model data structure**. Selecting this parameter can also cause the `rtModel` structure to be omitted completely from the generated code. When generating code for multiple integrated models, set this parameter the same for all of the models. Otherwise, the integrated application might exhibit unexpected behavior. For example, if you select the option in one model but not in another, the error status might not be registered by the integrated application. Do not select this parameter if you select the **MAT-file logging** option. The two options are incompatible.
Launch the Model Step Functions dialog box (see “Model Step Functions Dialog Box” on page 2-89) preview and modify the model’s *model*_step function prototype	Click **Configure Functions**. Based on the **Function specification** value you select for your *model*_step function (supported values include `Default model-step function` and `Model specific C prototype`), you can preview and modify the function prototype. Once you validate and apply your changes, you can generate code based on your function prototype modifications. For more information about using the **Configure Functions** button and the Model Step Functions dialog box, see “Controlling model_step Function Prototypes” on page 2-88.

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

Configuring a Model for Code Reuse

For ERT targets, you can configure how a model reuses code using the **Generate reusable code** parameter.

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

To...	Select...
Pass each root-level model input and output argument to the*`model`*`_step` function individually (the default)	**Generate reusable code** and **Pass root-level I/O as** > `Individual arguments`.
Pack root-level input arguments and root-level output arguments into separate structures that are then passed to the *`model`*`_step` function	**Generate reusable code** and **Pass root-level I/O as** > `Structure reference`

In some cases, selecting **Generate reusable code** can generate code that compiles but is not reentrant. For example, if any signal, `DWork` structure, or parameter data has a storage class other than `Auto`, global data structures are generated. To handle such cases, use the **Reusable code error diagnostic** parameter to choose the severity levels for diagnostics

In some cases, Real-Time Workshop Embedded Coder is unable to generate valid and compilable code. For example, if the model contains any of the following, the code generated would be invalid.

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

In these cases, the build terminates after reporting the problem.

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

Controlling Code Style

You can control the following style aspects in generated code:

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

For example, C code contains some syntactically required parentheses, and can contain additional parentheses that change semantics by overriding default operator precedence. C code can also contain optional parentheses that have no functional significance, but serve only to increase the readability of the code. Optional C parentheses vary between two stylistic extremes:

- Include the minimum parentheses required by C syntax and any precedence overrides, so that C precedence rules specify all semantics unless overridden by parentheses.
- Include the maximum parentheses that can exist without duplication, so that C precedence rules become irrelevant: parentheses alone completely specify all semantics.

Understanding code with minimum parentheses can require correctly applying nonobvious precedence rules, but maximum parentheses can hinder code reading by belaboring obvious precedence rules. Various parenthesization standards exist that specify one or the other extreme, or define an intermediate style that can be useful to human code readers.

You control the code style options by setting parameters on the **Real-Time Workshop > Code Style** pane. For details on the parameters, see "Real-Time Workshop Pane: Code Style" in the Real-Time Workshop Embedded Coder reference documentation.

Configuring Templates for Customizing Code

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

To...	Enter or Select...
Specify a template that defines the top-level organization and formatting of generated source code (`.c` or `.cpp`) files	Enter a code generation template (CGT) file for the **Source file (*.c) template** parameter. .
Specify a template that defines the top-level organization and formatting of generated header (`.h`) files	Enter a CGT file for the **Header file (*.h) template** parameter. This can be the same template file that you specify for **Source file (.c) template**, in which case identical banners are generated in source and header files. The default template is *`matlabroot`* `/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
Specify a template that organizes generated code into sections (such as includes, `typedefs`, functions, and more)	Enter a custom file processing (CFP) template file for the **File customization template** parameter. . A CFP template can emit code, directives, or comments into each section as required. See "Custom File Processing" on page 5-34 for detailed information.
Generate a model-specific example main program module	Select **Generate an example main program**. See "Generating the Main Program Module" on page 1-9 for more information.

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

For more detail, see the Module Packaging Features document. See also "Generating Custom File Banners" on page 5-55 for a simple example of how a code template can be applied to generate customized comment sections in generated code files.

Configuring the Placement of Data in Code

To...	Select or Enter...
Specify whether data is to be defined in the generated source file or in a single separate header file	Select **Auto**, **Data defined in source file**, or **Data defined in single separate source file** for the **Data definition** parameter.
Specify whether data is to be declared in the generated source file or in a single separate header file	Select **Auto**, **Data defined in source file**, or **Data defined in single separate source file** for the **Data declaration** parameter.
Specify the `#include` file delimiter to be used in generated files that contain the `#include` preprocessor directive for `mpt` data objects	Select **Auto**, **Data defined in source file**, or **Data defined in single separate source file** for the **#include file delimiter** parameter.
Name the generated module using the same name as the model or a user-specified name	Select **Not specified**, **Same as model**, or **User specified** for the **Module naming** parameter.
Control whether signal data objects are to be declared as global data in the generated code	Enter an integer value for the **Signal display level** parameter.
Declare a parameter data object as tunable global data in the generated code	Enter an integer value for the **Parameter tune level** parameter.

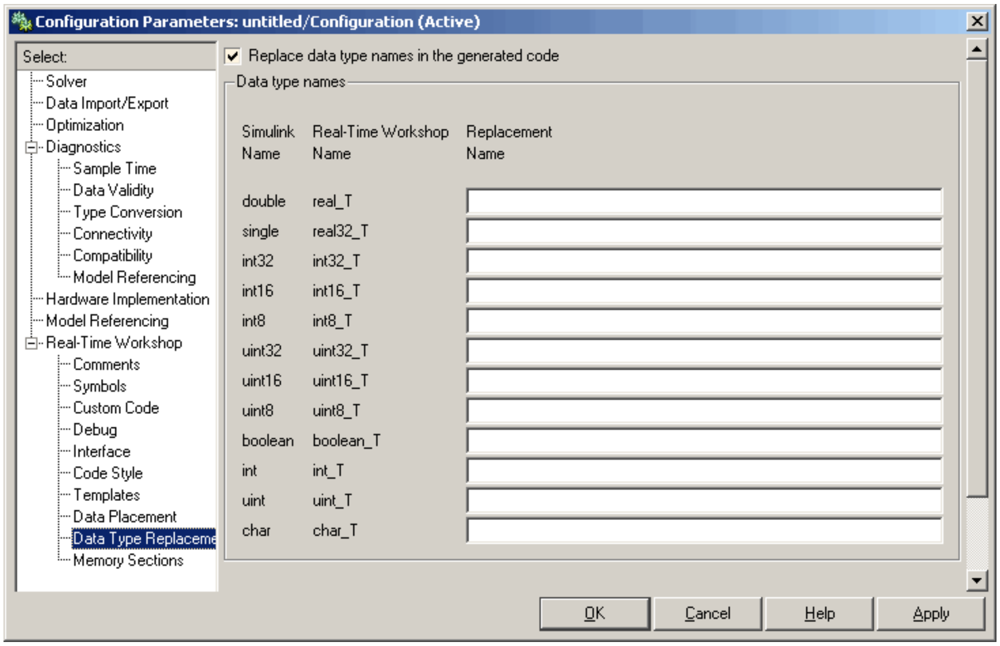
For details data placement, see the Module Packaging Features document.

Configuring Replacement Data Types

You can replace built-in data type names with user-defined replacement data type names in the generated code for a model.

To configure replacement data types,

1 Click **Replace data type names in the generated code**. A **Data type names** table appears. The table lists each Simulink built-in data type name with its corresponding Real-Time Workshop data type name.



☑ Replace data type names in the generated code

Data type names

Simulink Name	Real-Time Workshop Name	Replacement Name
double	real_T	
single	real32_T	
int32	int32_T	
int16	int16_T	
int8	int8_T	
uint32	uint32_T	
uint16	uint16_T	
uint8	uint8_T	
boolean	boolean_T	
int	int_T	
uint	uint_T	
char	char_T	

2 Selectively fill in fields in the third column with your replacement data types. Each replacement data type should be the name of a `Simulink.AliasType` object that exists in the base workspace. Replacements may be specified or not for each individual built-in type.

For each replacement data type you enter, the `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.

- For `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, and `boolean`, the replacement data type's `BaseType` must match the built-in data type.
- For `int`, `uint`, and `char`, the replacement data type's size must match the size displayed for `int` or `char` on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent. For more information, see "Replacing Built-In Data Type Names in Generated Code" in the Module Packaging Features document.

Configuring Memory Sections

You can configure a model such that the generated code includes comments and pragmas for

- Data defined in custom storage classes
- Internal data not defined in custom storage classes
- Model-level functions
- Atomic subsystem functions with or without separate data

To...	Select...
Specify the package that contains memory sections that you want to apply	The name of a package for **Package**. Click **Refresh package list** to refresh the list of available packages in your configuration.
Apply memory sections to initialize/start and terminate functions	A value for **Initialize/Terminate**.
Apply memory sections to step, run-time initialization, derivative, enable, and disable functions	A value for **Execution**.
Apply memory sections to constant parameters, constant, block I/O, or zero representation	A value for **Constants**.
Apply memory sections to root inputs or outputs	A value for **Inputs/Outputs**.
Apply memory sections to block I/O, Dwork vectors, run-time models, zero-crossings	A value for **Internal data**.
Apply memory sections to parameters	A value for **Parameters**.

The interface checks whether the specified package is on the MATLAB path and that the selected memory sections are in the package. The results of this validation appear in the field **Validation results**.

For details on using memory sections, see Chapter 4, “Memory Sections”.

Configuring Optimizations

To...	Select or Specify...
Control whether parameter data for reusable subsystems is generated in a separate header file for each subsystem or in a single parameter data structure	Select `Hierarchical` or `NonHierarchical` for **Parameter structure**.
Generate initialization code for root-level inports and outports with a value of zero	Select **Remove root level I/O zero initialization**.
Generate additional code to set float and double storage explicitly to value `0.0`	Select**Use memset to initialize floats and doubles to 0.0** When you set this parameter, the `memset` function clears internal storage (regardless of type) to the integer bit pattern `0` (that is, all bits are off). The additional code generated when the option is off, is slightly less efficient.If the representation of floating-point zero used by your compiler and target CPU is identical to the integer bit pattern `0`, you can gain efficiency by setting this parameter.
Suppress the generation of code that initializes internal work structures (for example, block states and block outputs) to zero	Select **Remove internal state zero initialization**.
Generate run-time initialization code for a block that has states only if the block is in a system that can reset its states, such as an enabled subsystem	Select **Optimize initialization code for model reference** This results in more efficient code, but requires that you not refer to the model from a Model block that resides in a system that resets its states. Such nesting results in an error. Turn this option off only if your application requires you refer to the model from Model blocks in systems that can reset their states.

To...	Select or Specify...
Remove code that ensures that execution of the generated code produces the same results as simulation when out-of-range conversions occur	Select **Remove code from floating-point to integer conversions that wraps out-of-range values**. This reduces the size and increases the speed of the generated code at the cost of potentially producing results that do not match simulation in the case of out-of-range values.
Suppress generation of code that guards against fixed-point division by zero	Select **Remove code that protects against division arithmetic exceptions**. When you select this parameter, simulation results and results from generated code may no longer be in bit-for-bit agreement.
To minimize the amount of memory allocated for absolute and elapsed time counters	Specify an integer value for **Application lifespan (days)** For more information on the allocation and operation of absolute and elapsed timers, see “Timing Services”, “Using Timers in Asynchronous Tasks”, and “Controlling Memory Allocation for Time Counters” in the Real-Time Workshop documentation.

Tips for Optimizing the Generated Code

In this section...

Introduction

Real-Time Workshop Embedded Coder features a number of code generation options that can help you further optimize the generated code. This section highlights code generation options you can use to improve performance and reduce code size.

Most of the tips in this section apply specifically to the ERT target. See also the "Optimizing a Model for Code Generation" section of the Real-Time Workshop documentation for optimization techniques that are common to all target configurations.

Using Auto-Optimized Targets

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

Before generating and deploying code, consider using one of these optimized target variants. The optimized ERT target variants are discussed in detail in "Generating Efficient Code with Optimized ERT Targets" on page 5-26.

Using Configuration Wizard Blocks

Real-Time Workshop Embedded Coder provides a library of *Configuration Wizard* blocks and scripts to help you configure and optimize code generation from your models quickly and easily.

When you add one of the preset Configuration Wizard blocks to your model and double-click it, an M-file script executes and configures all parameters of the model's active configuration set without user intervention. The preset blocks configure the options optimally for common fixed- and floating-point code generation scenarios.

You can also create custom Configuration Wizard scripts and blocks.

See "Optimizing Your Model with Configuration Wizard Blocks and Scripts" on page 5-61 for detailed information.

Setting Hardware Implementation Parameters Correctly

Correct specification of target-specific characteristics of generated code (such as word sizes for `char`, `short`, `int`, and `long` data types, or desired rounding behaviors in integer operations) can be critical in embedded systems development. The **Hardware Implementation** category of options in a configuration set provides a simple and flexible way to control such characteristics in both simulation and code generation.

Before generating and deploying code, you should become familiar with the options on the **Hardware Implementation** pane of the Configuration Parameters dialog box. See "Hardware Implementation Pane" in the Simulink documentation and "Configuring Hardware Properties and Constraints" in the Real-Time Workshop documentation for full details on the **Hardware Implementation** pane.

By configuring the **Hardware Implementation** properties of your model's active configuration set to match the behaviors of your compiler and hardware, you can generate more efficient code. For example, if you specify the **Byte ordering** property, you can avoid generation of extra code that tests the byte ordering of the target CPU.

You can use the `rtwdemo_targetsettings` demo model to determine some implementation-dependent characteristics of your C or C++ compiler, as well as characteristics of your target hardware. By using this model in conjunction with your target development system and debugger, you can observe the behavior of the code as it executes on the target. You can then use this information to configure the **Hardware Implementation** parameters of your model.

To use this model, type the command

```
rtwdemo_targetsettings
```

Follow the instructions in the model window.

Removing Unnecessary Initialization Code

Consider selecting the **Remove internal state zero initialization** and **Remove root level I/O zero initialization** options on the **Optimization** pane.

These options (both off by default) control whether internal data (block states and block outputs) and external data (root inports and outports whose value is zero) are initialized. Initializing the internal and external data whose value is zero is a precaution and may not be necessary for your application. Many embedded application environments initialize all RAM to zero at startup, making generation of initialization code redundant.

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

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

In cases where you have turned on **Remove internal state zero initialization** but still want to get the same answer on every run from a Real-Time Workshop Embedded Coder generated S-function, you can use either of the following MATLAB commands before each run:

```
clear SFcnName
```

where *SFcnName* is the name of the S-function, or

```
clear mex
```

A related option, **Use memset to initialize floats and doubles**, lets you control the representation of zero used during initialization. See “Use memset to initialize floats and doubles to 0.0” in the Simulink reference documentation.

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

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

Generating Pure Integer Code If Possible

If your application uses only integer arithmetic, deselect the **Support floating-point numbers** option in the **Software environment** section of the **Interface** pane to ensure that generated code contains no floating-point data or operations. When this option is deselected, an error is raised if any noninteger data or expressions are encountered during code generation. The error message reports the offending blocks and parameters.

Disabling MAT-File Logging

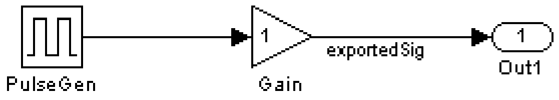
Clear the **MAT-file logging** option in the **Verification** section of the **Interface** pane. This setting is the default, and is recommended for embedded applications because it eliminates the extra code and memory usage required to initialize, update, and clean up logging variables. In addition to these efficiencies, clearing the **MAT-file logging** option lets you exploit further efficiencies under certain conditions. See "Using Virtualized Output Ports Optimization" on page 2-39 for information.

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

Using Virtualized Output Ports Optimization

The *virtualized output ports* optimization lets you store the signal entering the root output port as a global variable. This eliminates code and data storage associated with root output ports when the **MAT-file logging** option is cleared and the TLC variable `FullRootOutputVector` equals `0`, both of which are defaults for Real-Time Workshop Embedded Coder.

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



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

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

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

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

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

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

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

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

```
%assign FullRootOutputVector = 1
```

to the Real-Time Workshop Embedded Coder system target file. Alternatively, you can enter the assignment with **TLC options** on the **Real-Time Workshop** pane of the Configuration Parameters dialog box.

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

Using Stack Space Allocation Options

Real-Time Workshop offers a number of options that let you control how signals in your model are stored and represented in the generated code. This section discusses options that

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

For a complete discussion of signal storage options, see the "Working with Data Structures" section of the Real-Time Workshop documentation.

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

1. Select the **Optimization** tab of the Configuration Parameters dialog box. Make sure that the **Signal storage reuse** option is selected. If **Signal storage reuse** is off, the **Enable local block outputs** option is not available.
2. Select the **Enable local block outputs** option. Click **Apply** if necessary.

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

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

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

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

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

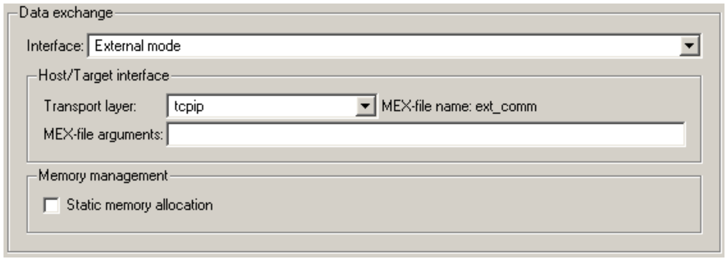
```
%assign MaxStackSize = 4096
```

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

Using External Mode with the ERT Target

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

This section discusses external mode options that may be of special interest to embedded systems designers. The next figure shows the **Data Exchange** subpane of the Configuration Parameters dialog box, **Interface** pane, with `External mode` selected.



Data exchange
Interface: External mode
Host/Target interface
Transport layer: tcpip MEX-file name: ext_comm
MEX-file arguments:
Memory management
☐ Static memory allocation

Memory Management

Consider the **Memory management** option **Static memory allocation** before generating external mode code for an embedded target. Static memory allocation is generally desirable, as it reduces overhead and promotes deterministic performance.

When you select the **Static memory allocation** option, static external mode communication buffers are allocated in the target application. When **Static memory allocation** is deselected, communication buffers are allocated dynamically (with `malloc`) at run time.

Generation of Pure Integer Code with External Mode

Real-Time Workshop Embedded Coder supports generation of pure integer code when external mode code is generated. To do this, select the **External mode** option, and deselect the **Support floating-point numbers** option in the **Software environment** section of the **Interface** pane.

This enhancement lets you generate external mode code that is free of any storage definitions of double or float data type, and allows your code to run on integer-only processors

If you intend to generate pure integer code with **External mode** on, note the following requirements:

- All trigger signals must be of data type `int32`. Use a Data Type Conversion block if needed.
- When pure integer code is generated, the simulation stop time specified in the **Solver** options is ignored. To specify a stop time, run your target application from the MATLAB command line and use the `-tf` option. (See "Running the External Program" in the "External Mode" section of the Real-Time Workshop documentation.) If you do not specify this option, the application executes indefinitely (as if the stop time were `inf`).

  When executing pure integer target applications, the stop time specified by the `-tf` command line option is interpreted as the number of base rate ticks to execute, rather than as an elapsed time in seconds. The number of ticks is computed as

  ```
  stop time in seconds / base rate step size in seconds
  ```

Generating and Using an HTML Code Generation Report

In this section...

Overview

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

- The Generated Source Files section of the Contents pane contains a table of source code files generated from your model. You can view the source code in a MATLAB Web browser window.

  If you selected the traceability option **Code-to-block highlighting**, hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click on the hyperlinks to view the relevant blocks or subsystems in a Simulink model window.

  If you selected the traceability option **Block-to-code highlighting**, traceability support lets you view the generated code for any block in the model. To highlight a block's generated code in the HTML report, right-click the block and select **Real-Time Workshop > Highlight Code**.

- The Summary section lists version and date information, TLC options used in code generation, and Simulink model settings. The Configuration Settings at the Time of Code Generation link opens a noneditable view of the Configuration Parameters dialog that shows all of the settings at the time of code generation.

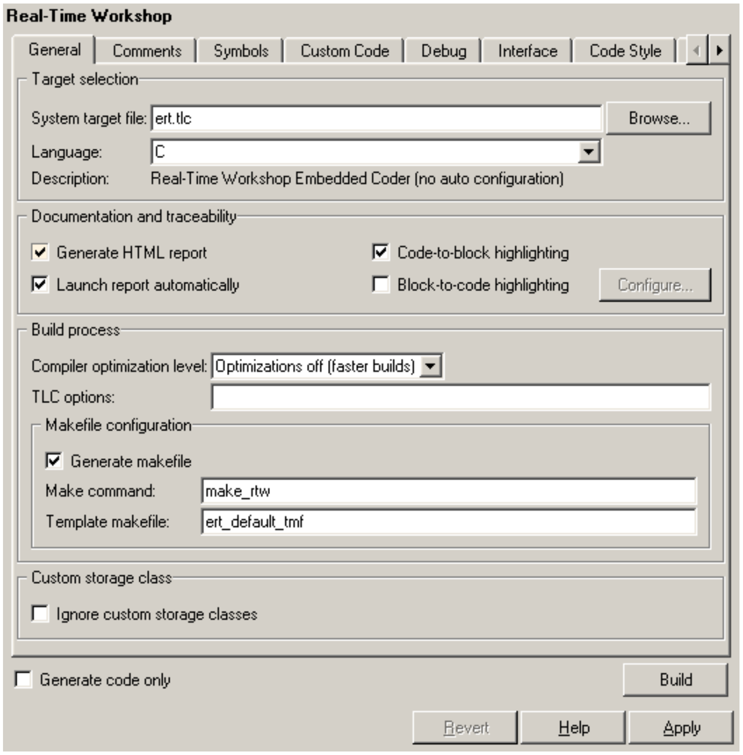
- The Traceability Report section appears only if you selected the traceability option **Block-to-code highlighting**. It lists **Eliminated / Virtual Blocks** versus **Traceable Blocks**, helping to provide a complete mapping between your model blocks and your generated code.
- The Subsystem Report section contains information on nonvirtual subsystems in the model.

Generating an HTML Code Generation Report

To generate a code generation report,

1. Open the Configuration Parameters dialog box or Model Explorer and navigate to the **Real-Time Workshop** pane.
2. In the **Documentation and traceability** subpane, select **Generate HTML report**. By default, **Launch report automatically** and **Code-to-block highlighting** also are selected, and **Block-to-code highlighting** is cleared, as shown in the figure below.

   You can select or deselect any of these options as desired.



Target selection

System target file: ert.tlc

Language: C

Description: Real-Time Workshop Embedded Coder (no auto configuration)

Documentation and traceability

[✓] Generate HTML report

[✓] Code-to-block highlighting

[✓] Launch report automatically

[ ] Block-to-code highlighting

Build process

Compiler optimization level: Optimizations off (faster builds)

TLC options:

Makefile configuration

[✓] Generate makefile

Make command: make_rtw

Template makefile: ert_default_tmf

Custom storage class

[ ] Ignore custom storage classes

[ ] Generate code only

3 Generate code from your model or subsystem (for example, for a model, by clicking **Build** on the **Real-Time Workshop** pane of the Configuration Parameters dialog box).

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

5 If you selected **Launch report automatically**, Real-Time Workshop automatically opens a MATLAB Web browser window and displays the code generation report.

If you did not select **Launch report automatically**, you can open the code generation report (*`model`*`_codegen_rpt.html` or *`subsystem`*`_codegen_rpt.html`) manually into a MATLAB Web browser window, or into another Web browser.

6 If you selected **Code-to-block highlighting**, hyperlinks to blocks in the generating model are created in the report files. When you view the report files in MATLAB, clicking on these hyperlinks displays and highlights the referenced blocks in the model. For more information, see “Using Code-to-Block Highlighting” on page 2-47.

7 If you selected **Block-to-code highlighting**, block-to-code highlighting support is included in the generated HTML report. To highlight the generated code for a block in your Simulink model, right-click the block and select **Real-Time Workshop > Highlight Code**. This selection highlights the generated code for the block in the HTML code generation report. For more information, see “Using Block-to-Code Highlighting” on page 2-49 and “Viewing the Traceability Report” on page 2-52.

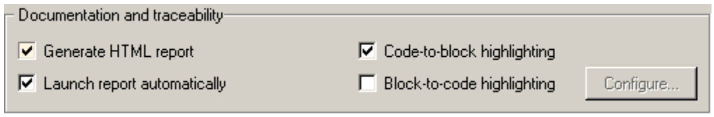
Notes

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

Using Code-to-Block Highlighting

To use **Code-to-block highlighting**,

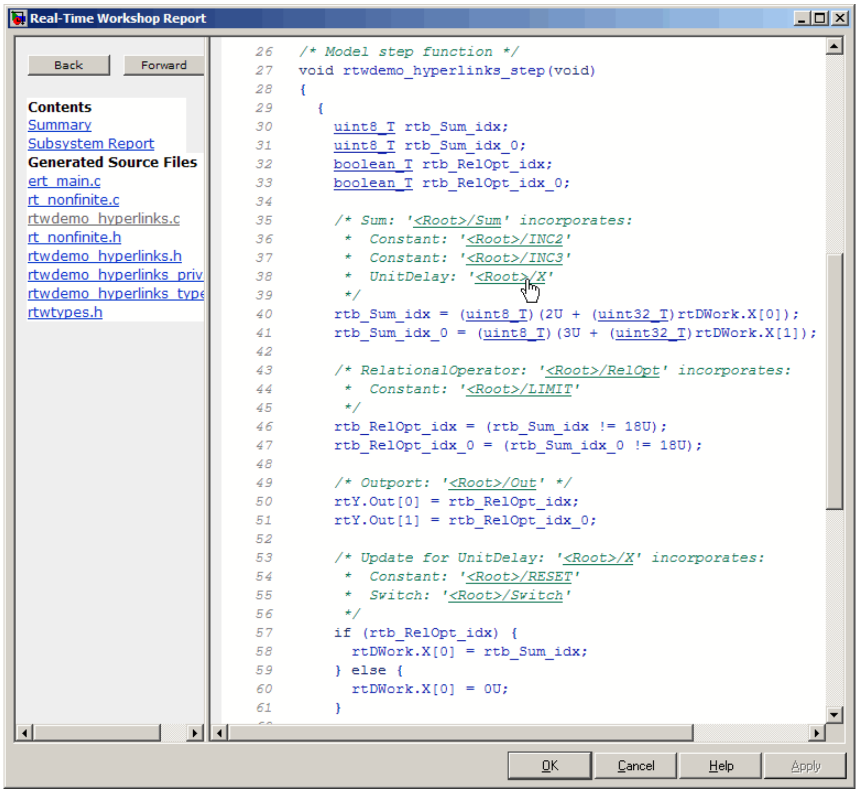
1 Open an ERT-based model and go to the **Real-Time Workshop** pane of the Configuration Parameters dialog box. Select the option **Generate HTML report**. By default, **Launch report automatically** and **Code-to-block highlighting** also are selected.


Documentation and traceability

[✓] Generate HTML report
[✓] Launch report automatically
[✓] Code-to-block highlighting
[ ] Block-to-code highlighting

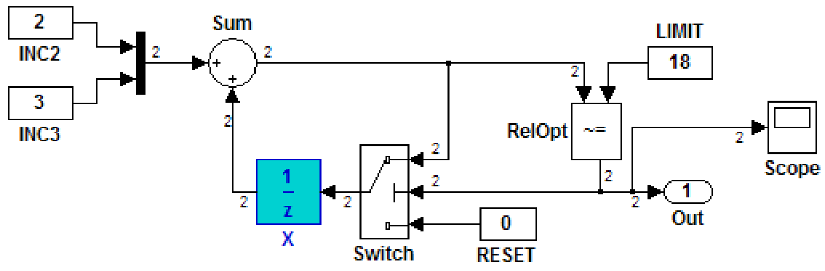
2 Build your model. This will launch an HTML code generation report.

3 In the HTML report window, click any of the hyperlinks present to highlight the source block. For example, in the HTML report shown below for the demo model `rtwdemo_hyperlinks`, you could click the first UnitDelay hyperlink in the generated code for the model step function.



```
/* Model step function */
void rtwdemo_hyperlinks_step(void)
{
  {
    uint8_T rtb_Sum_idx;
    uint8_T rtb_Sum_idx_0;
    boolean_T rtb_RelOpt_idx;
    boolean_T rtb_RelOpt_idx_0;

    /* Sum: '<Root>/Sum' incorporates:
     *  Constant: '<Root>/INC2'
     *  Constant: '<Root>/INC3'
     *  UnitDelay: '<Root>/X'
     */
    rtb_Sum_idx = (uint8_T)(2U + (uint32_T)rtDWork.X[0]);
    rtb_Sum_idx_0 = (uint8_T)(3U + (uint32_T)rtDWork.X[1]);

    /* RelationalOperator: '<Root>/RelOpt' incorporates:
     *  Constant: '<Root>/LIMIT'
     */
    rtb_RelOpt_idx = (rtb_Sum_idx != 18U);
    rtb_RelOpt_idx_0 = (rtb_Sum_idx_0 != 18U);

    /* Outport: '<Root>/Out' */
    rtY.Out[0] = rtb_RelOpt_idx;
    rtY.Out[1] = rtb_RelOpt_idx_0;

    /* Update for UnitDelay: '<Root>/X' incorporates:
     *  Constant: '<Root>/RESET'
     *  Switch: '<Root>/Switch'
     */
    if (rtb_RelOpt_idx) {
      rtDWork.X[0] = rtb_Sum_idx;
    } else {
      rtDWork.X[0] = 0U;
    }
```

4 Making this selection highlights the corresponding source block in the Simulink model window.

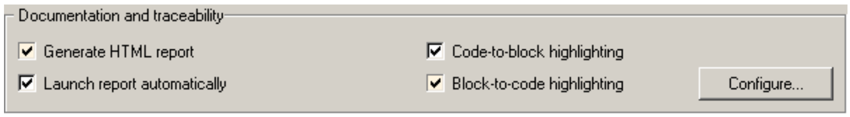


See also the demo `rtwdemo_hyperlinks`, which walks you through using **Code-to-block highlighting** .

Using Block-to-Code Highlighting

To use **Block-to-code highlighting**,

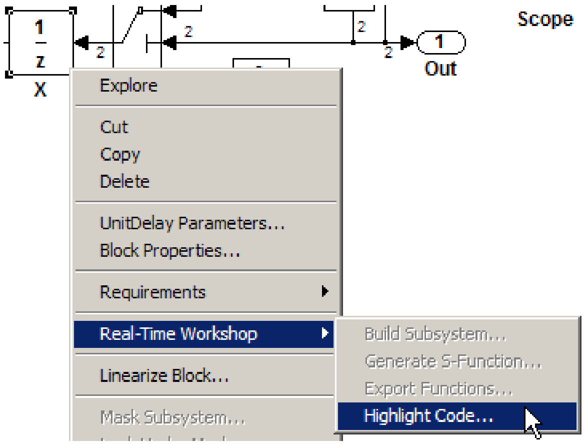
1 Open an ERT-based model and go to the **Real-Time Workshop** pane of the Configuration Parameters dialog box. Select the options **Generate HTML report**, **Launch report automatically**, and **Block-to-code highlighting**. (Selecting **Block-to-code highlighting** also enables the **Configure** button, which you can use to open the Block-to-code highlighting dialog box; see "Using the Block-to-Code Highlighting Dialog Box to Load Existing Trace Information" on page 2-51.)


Documentation and traceability

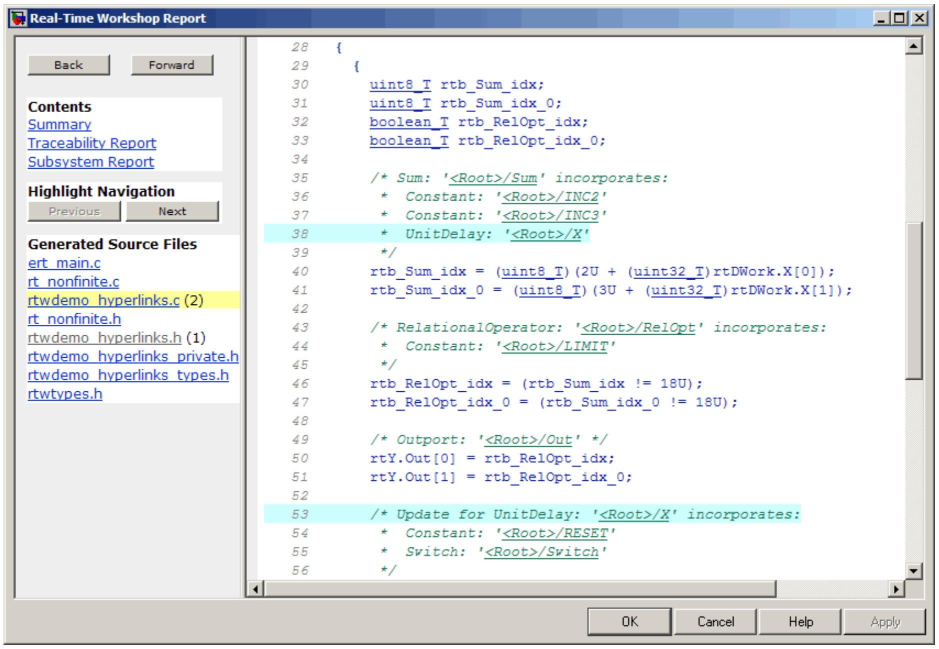
✓ Generate HTML report
✓ Launch report automatically
✓ Code-to-block highlighting
✓ Block-to-code highlighting

2 Build your model. This will launch an HTML code generation report.

3 In the model window, right-click any block. In the right-click menu, select **Real-Time Workshop > Highlight Code**.



4 This selection highlights the generated code for the block in the HTML code generation report. The total number of highlighted lines is displayed next to each source file name in the left panel of the HTML report. **Previous** and **Next** buttons help you navigate through the highlighted lines.



```
{
  {
    uint8_T rtb_Sum_idx;
    uint8_T rtb_Sum_idx_0;
    boolean_T rtb_RelOpt_idx;
    boolean_T rtb_RelOpt_idx_0;

    /* Sum: '<Root>/Sum' incorporates:
     *  Constant: '<Root>/INC2'
     *  Constant: '<Root>/INC3'
     *  UnitDelay: '<Root>/X'
     */
    rtb_Sum_idx = (uint8_T)(2U + (uint32_T)rtDWork.X[0]);
    rtb_Sum_idx_0 = (uint8_T)(3U + (uint32_T)rtDWork.X[1]);

    /* RelationalOperator: '<Root>/RelOpt' incorporates:
     *  Constant: '<Root>/LIMIT'
     */
    rtb_RelOpt_idx = (rtb_Sum_idx != 18U);
    rtb_RelOpt_idx_0 = (rtb_Sum_idx_0 != 18U);

    /* Outport: '<Root>/Out' */
    rtY.Out[0] = rtb_RelOpt_idx;
    rtY.Out[1] = rtb_RelOpt_idx_0;

    /* Update for UnitDelay: '<Root>/X' incorporates:
     *  Constant: '<Root>/RESET'
     *  Switch: '<Root>/Switch'
     */
```

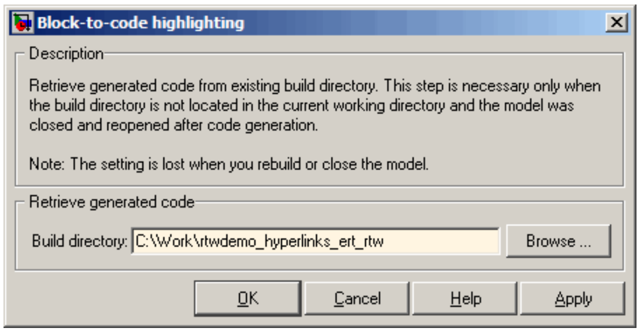
See also the demo `rtwdemo_hyperlinks`, which walks you through using **Block-to-code highlighting**.

Note If you later close and reopen your model, you may find that you cannot use right-click/**Real-Time Workshop > Highlight Code** to trace a block's code, because **Highlight Code** is greyed out. This means that a build directory for your model cannot be found in the current working directory. To address this you can do any of the following:

- Reset the current working directory to the parent directory of the existing build directory.
- Select **Block-to-code highlighting** and rebuild the model. This regenerates the build directory into the current working directory.
- Click the **Configure** button and reload the model's trace information using the Block-to-code highlighting dialog box. See "Using the Block-to-Code Highlighting Dialog Box to Load Existing Trace Information" on page 2-51.

Using the Block-to-Code Highlighting Dialog Box to Load Existing Trace Information

To open the Block-to-code highlighting dialog box, click the **Configure** button on the **Documentation and traceability** subpane. You can use this dialog box to reconnect your model with a previously-generated build directory, including trace information for block-to-code highlighting.



Block-to-code highlighting

Description

Retrieve generated code from existing build directory. This step is necessary only when the build directory is not located in the current working directory and the model was closed and reopened after code generation.

Note: The setting is lost when you rebuild or close the model.

Retrieve generated code

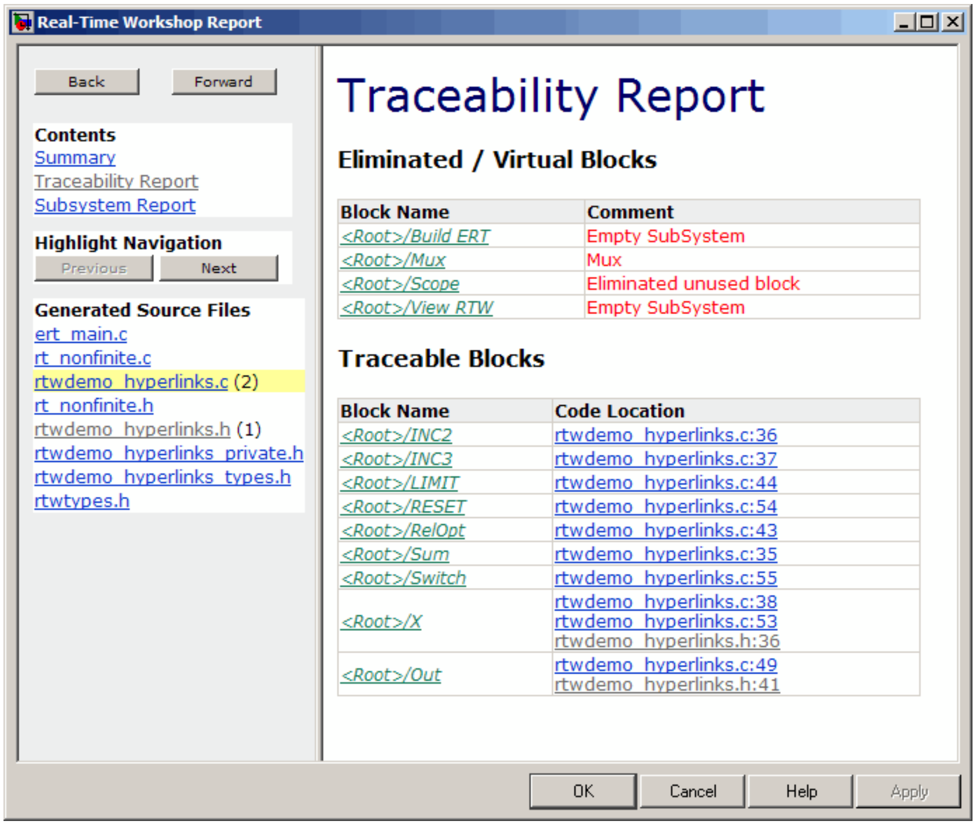
Build directory: C:\Work\rtwdemo_hyperlinks_ert_rtw

If you find that you cannot use right-click/**Real-Time Workshop > Highlight Code** to trace a block's code, because **Highlight Code** is greyed out, it means that a build directory for your model cannot be found in the current working directory. To fix this without having to reset the current working directory or rebuild the model,

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

Viewing the Traceability Report

When you select the **Block-to-code highlighting** parameter discussed in the previous section, the generated HTML report contains a traceability report. The traceability report contains sections that allow you to account for **Eliminated / Virtual Blocks** versus **Traceable Blocks**, providing a complete mapping between blocks and code.



Traceability Report

Eliminated / Virtual Blocks

Block Name	Comment
<Root>/Build ERT	Empty SubSystem
<Root>/Mux	Mux
<Root>/Scope	Eliminated unused block
<Root>/View RTW	Empty SubSystem

Traceable Blocks

Block Name	Code Location
<Root>/INC2	rtwdemo_hyperlinks.c:36
<Root>/INC3	rtwdemo_hyperlinks.c:37
<Root>/LIMIT	rtwdemo_hyperlinks.c:44
<Root>/RESET	rtwdemo_hyperlinks.c:54
<Root>/RelOpt	rtwdemo_hyperlinks.c:43
<Root>/Sum	rtwdemo_hyperlinks.c:35
<Root>/Switch	rtwdemo_hyperlinks.c:55
<Root>/X	rtwdemo_hyperlinks.c:38 rtwdemo_hyperlinks.c:53 rtwdemo_hyperlinks.h:36
<Root>/Out	rtwdemo_hyperlinks.c:49 rtwdemo_hyperlinks.h:41

See the demo `rtwdemo_hyperlinks`, which walks you through using the traceability report.

Traceability Limitations

The following limitations apply to the traceability features of HTML code generation reports.

- If a block name in your model contains a single quote (`'`), code-to-block highlighting and block-to-code highlighting are disabled for that block.
- If an asterisk (`*`) in a block name in your model causes a name-mangling ambiguity relative to other names in the model, code-to-block highlighting and block-to-code highlighting are disabled for that block. This is most

likely to occur if an asterisk precedes or follows a slash (`/`) in a block name or appears at the end of a block name.

- If a block name in your model contains the character (`char(255)`), code-to-block highlighting and block-to-code highlighting are disabled for that block.
- Some types of subsystems are not traceable using **Block-to-code highlighting** at the subsystem block level:
  - Virtual subsystems
  - Masked subsystems
  - Nonvirtual subsystems for which code has been optimized away

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

Generating Code Within MISRA-C Guidelines

The Motor Industry Software Reliability Association (MISRA) has established "Guidelines for the Use of the C Language in Critical Systems" (MISRA-C). For general information about MISRA-C, see `www.misra-c.com`.

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

Automatic S-Function Wrapper Generation

In this section...

Overview

An S-function wrapper is an S-function that calls your C or C++ code from within Simulink. S-function wrappers provide a standard interface between Simulink and externally written code, allowing you to integrate your code into a model with minimal modification. This is useful for software-in-the-loop (SIL) code verification (validating your generated code in Simulink), as well as for simulation acceleration purposes. For a complete description of wrapper S-functions, see the Simulink Writing S-Functions document.

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

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

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

The build process then compiles and links `*model*_sf.c` or `.cpp` with `*model*.c` or `.cpp` and the other Real-Time Workshop Embedded Coder generated code modules, building a MEX-file. The MEX-file is named `*model*_sf.*mexext*`. (`*mexext*` is the file extension for MEX-files on your platform, as given by the MATLAB `mexext` command.) The MEX-file is stored in your working directory. Finally, Real-Time Workshop creates and opens an untitled model containing the generated S-Function block.

Note To generate a wrapper S-function for a subsystem, you can use a right-click subsystem build. Right-click the subsystem block in your model, select **Real-Time Workshop > Generate S-Function**, and in the Generate S-Function dialog box, select **Use Embedded Coder** and click **Build**.

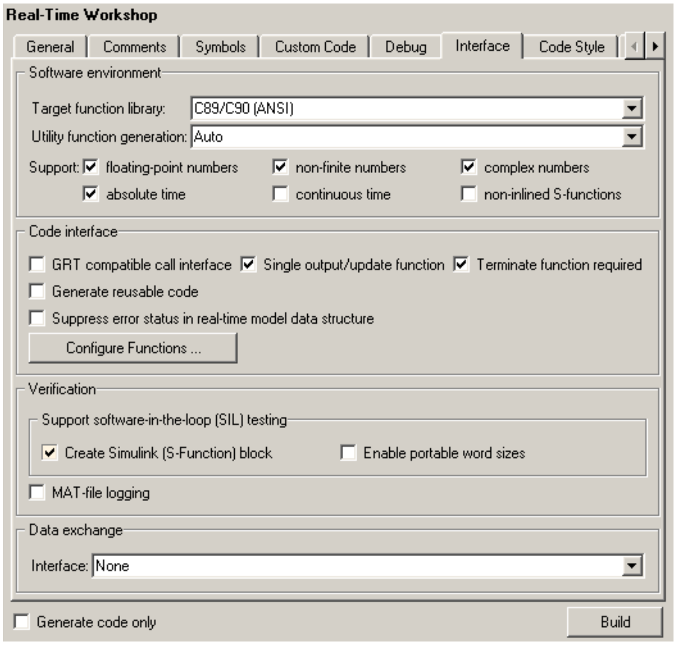
Generating an ERT S-Function Wrapper

To generate an S-function wrapper for your Real-Time Workshop Embedded Coder code, open your ERT-based Simulink model and do the following:

1. Open the Configuration Parameters dialog box.

2. Select the **Interface** pane.

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



Real-Time Workshop

General | Comments | Symbols | Custom Code | Debug | Interface | Code Style

Software environment

Target function library: C89/C90 (ANSI)

Utility function generation: Auto

Support: [✓] floating-point numbers [✓] non-finite numbers [✓] complex numbers [✓] absolute time [ ] continuous time [ ] non-inlined S-functions

Code interface

[ ] GRT compatible call interface [✓] Single output/update function [✓] Terminate function required

[ ] Generate reusable code

[ ] Suppress error status in real-time model data structure

Configure Functions ...

Verification

Support software-in-the-loop (SIL) testing

[✓] Create Simulink (S-Function) block [ ] Enable portable word sizes

[ ] MAT-file logging

Data exchange

Interface: None

[ ] Generate code only

Build

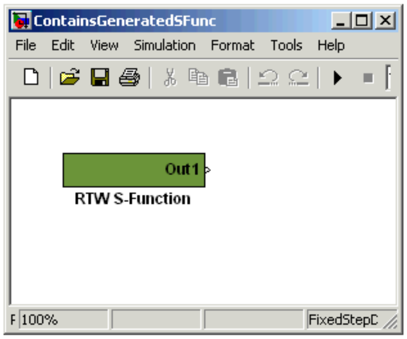
4 Configure the other code generation options as required.

5 To ensure that memory for the S-Function is initialized to zero, you should deselect the following options in the **Data Initialization** subpane of the **Optimization** pane:

- **"Remove root level I/O zero initialization"**
- **"Remove internal state zero initialization"**
- **"Use memset to initialize floats and doubles to 0.0"**

6 Select the **Real-Time Workshop** pane and click the **Build** button.

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



8 Save the new model.

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

S-Function Wrapper Generation Limitations

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

- Continuous sample time is not supported. The **Support continuous time** option should not be selected when generating a Real-Time Workshop Embedded Coder S-function wrapper.
- Models that contain S-function blocks for which the S-function is not inlined with a TLC file are not supported when generating a Real-Time Workshop Embedded Coder S-function wrapper.
- You cannot use multiple instances of a Real-Time Workshop Embedded Coder generated S-function block within a model, because the code uses static memory allocation. Each instance potentially can overwrite global data values of the others.
- Real-Time Workshop Embedded Coder S-function wrappers can be used with other blocks and models for such purposes as SIL code verification and simulation acceleration, but cannot be used for code generation.
- A MEX S-function wrapper must only be used in the version of MATLAB in which the wrapper is created.

Verifying Generated Code with Software-in-the-loop Testing

In this section...

Overview

Real-Time Workshop Embedded Coder provides software-in-the-loop (SIL) code verification for subsystems using ERT S-function wrappers, described in "Automatic S-Function Wrapper Generation" on page 2-56. When processor word sizes differ between host and target platforms (for example, a 32-bit host and a 16-bit target), there are two ways to configure your Simulink model to simulate target behavior on the MATLAB host with SIL:

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

Select the *hardware emulation* method if you need the MATLAB host computer to simulate the bit-true behavior of the generated code on the target deployment system. In this case, the code that you generate for simulation on the MATLAB host might contain additional code, such as data type casts, that is necessary to ensure behavior consistent with the target environment. (See also "Configuring Optimizations" in the Real-Time Workshop documentation for settings in the **Code generation** subpane of the **Optimization** pane that affect the generated code.) After SIL testing on the MATLAB host, you must select **None** for **Emulation hardware** and then regenerate code for the target before deployment.

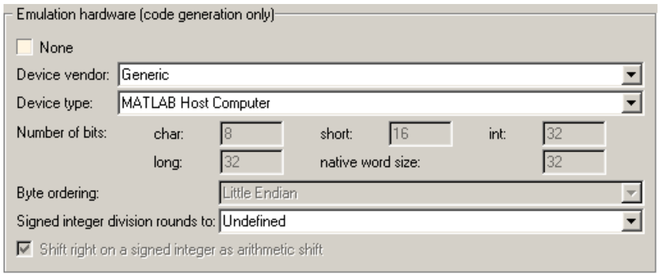
Select the *portable word sizes* method if you want to generate code that can be compiled without alteration both for SIL testing on the MATLAB host computer and deployment on the target system. In this case, the code that you generate has conditional processing macros that allow you to first compile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

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

Validating Generated Code on the MATLAB Host Computer Using Hardware Emulation

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

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


Emulation hardware (code generation only)

☐ None

Device vendor: Generic

Device type: MATLAB Host Computer

Number of bits: char: 8 short: 16 int: 32

long: 32 native word size: 32

Byte ordering: Little Endian

Signed integer division rounds to: Undefined

☑ Shift right on a signed integer as arithmetic shift

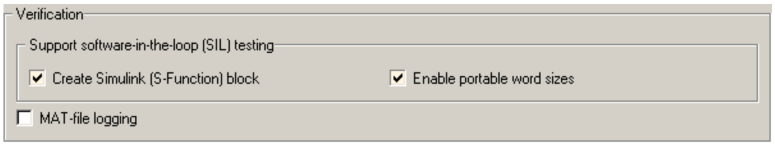
You can then right-click the subsystem that you want to test on the MATLAB host, and select **Real-Time Workshop > Build Subsystem** to build it. This will generate an S-function wrapper for the generated subsystem code, which can be used on the host to verify that the generated code provides the same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

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

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

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

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


Verification
Support software-in-the-loop (SIL) testing
[✓] Create Simulink (S-Function) block
[✓] Enable portable word sizes
[ ] MAT-file logging

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

```
#ifdef PORTABLE_WORDSIZES    /* PORTABLE_WORDSIZES defined */
# include "tmwtypes.h"
```

```
#else                          /* PORTABLE_WORDSIZES not defined */
#define __TMWTYPES__
#include <limits.h>
...
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef int int16_T;
typedef unsigned int uint16_T;
typedef long int32_T;
typedef unsigned long uint32_T;
typedef float real32_T;
typedef double real64_T;
...
#endif                         /* PORTABLE_WORDSIZES */
```

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

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

Portable Word Sizes Limitations

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

- Numerical results of the S-function simulation on the MATLAB host may differ from results on the actual target due to differences in target characteristics, such as
  - C integral promotion in expressions may be different on the target processor
  - Signed integer division rounding behavior may be different on the target processor
  - Signed integer arithmetic shift right may behave differently on the target processor
  - Floating-point precision may be different on the target processor

Exporting Function-Call Subsystems

In this section...

Overview

Real-Time Workshop Embedded Coder provides code export capabilities that you can use to

- Automatically generate code for
  - A function-call subsystem that contains only blocks that support code generation
  - A virtual subsystem that contains only such subsystems and a few other types of blocks
- Optionally generate an ERT S-function wrapper for the generated code

You can use these capabilities only if the subsystem and its interface to the Simulink model conform to certain requirements and constraints, as described in "Requirements for Exporting Function-Call Subsystems" on page 2-65. For limitations that apply, see "Function-Call Subsystems Export Limitations" on page 2-73.

Exported Subsystems Demo

To see a demo of exported function-call subsystems, type `rtwdemo_export_functions` in the MATLAB Command Window.

Additional Information

See the following in the Simulink documentation for additional information relating to exporting function-call subsystems:

- "Systems and Subsystems"
- "Signals"
- "Triggered Subsystems"
- "Function-Call Subsystems"
- *Writing S-Functions*

If you want to use Stateflow blocks to trigger exportable function-call subsystems, you may also need information from the *Stateflow® and Stateflow® Coder™ User's Guide*.

Requirements for Exporting Function-Call Subsystems

To be exportable as code, a function-call subsystem, or a virtual subsystem that contains such subsystems, must meet certain requirements. Most requirements are similar for either type of export, but some apply only to virtual subsystems. The requirements that affect all Simulink code generation also apply.

For brevity, *exported subsystem* in this section means only an exported function-call subsystem or an exported virtual subsystem that contains such subsystems. The requirements listed do not necessarily apply to other types of exported subsystems.

Requirements for All Exported Subsystems

These requirements apply to both exported function-call subsystems and exported virtual subsystems that contain such subsystems.

Blocks Must Support Code Generation. All blocks within an exported subsystem must support code generation. However, blocks outside the subsystem need not support code generation unless they will be converted to code in some other context.

Blocks Must Not Use Absolute Time. Certain blocks use absolute time. Blocks that use absolute time are not supported in exported function-call subsystems. For a complete list of such blocks, see "Blocks That Depend on Absolute Time" in the Real-Time Workshop documentation.

Blocks Must Not Depend on Elapsed Time. Certain blocks, like the Sine Wave block and Discrete Integrator block, depend on elapsed time. If an exported function-call subsystem contains any blocks that depend on elapsed time, the subsystem must specify periodic execution. See "Exporting Function-Call Subsystems That Depend on Elapsed Time" on page 2-68 in the Real-Time Workshop documentation.

Trigger Signals Require a Common Source. If more than one trigger signal crosses the boundary of an exported system, all of the trigger signals must be periodic and originate from the same function-call initiator.

Trigger Signals Must Be Scalar. A trigger signal that crosses the boundary of an exported subsystem must be scalar. Input and output data signals that do not act as triggers need not be scalar.

Data Signals Must Be Nonvirtual. A data signal that crosses the boundary of an exported system cannot be a virtual bus, and cannot be implemented as a `Goto-From` connection. Every data signal crossing the export boundary must be scalar, muxed, or a nonvirtual bus.

Requirements for Exported Virtual Subsystems

These requirements apply only to exported virtual subsystems that contain function-call subsystems.

Virtual Subsystem Must Use Only Permissible Blocks. The top level of an exported virtual subsystem that contains function-call subsystem blocks can contain only the following other types of blocks:

- Input and Output blocks (ports)

- Constant blocks (including blocks that resolve to constants, such as Add)
- Merge blocks
- Virtual connection blocks (Mux, Demux, Bus Creator, Bus Selector, Signal Specification)
- Signal-viewer blocks, such as `Scope` blocks

These restrictions do *not* apply within function-call subsystems, whether or not they appear in a virtual subsystem. They apply only at the top level of an exported virtual subsystem that contains one or more function-call subsystems.

Constant Blocks Must Be Inlined. When a constant block appears at the top level of an exported virtual subsystem, the containing model must check `Inline parameters` on the Optimization pane of the Configuration Parameters dialog box.

Constant Outputs Must Specify a Storage Class. When a constant signal drives an output port of an exported virtual subsystem, the signal must specify a storage class.

Techniques for Exporting Function-Call Subsystems

To export a function-call subsystem, or a virtual subsystem that contains function-call subsystems,

1. Ensure that the subsystem to be exported satisfies the "Requirements for Exporting Function-Call Subsystems" on page 2-65.
2. In the Configuration Parameters dialog box:
   a. On the **Real-Time Workshop** pane, specify an ERT code generation target such as `ert.tlc`.
   b. If you want an ERT S-function wrapper for the generated code, go to the **Interface** pane and select **Create Simulink (S-function) block**.
   c. Click **OK** or **Apply**.
3. Right-click the subsystem block and choose **Real-Time Workshop > Export Functions** from the context menu.

The `Build code for subsystem:` *`Subsystem`* dialog box appears. This dialog box is not specific to exporting function-call subsystems, and generating code does not require entering information in the box.

4 Click **Build**.

The MATLAB Command Window displays messages similar to those for any code generation sequence. Simulink generates code and places it in the working directory.

If you checked **Create Simulink (S-function) block** in step 2b, Simulink opens a new window that contains an S-function block that represents the generated code. This block has the same size, shape, and connectors as the original subsystem.

Code generation and optional block creation are now complete. You can test and use the code and optional block as you could any generated ERT code and S-function block.

Optimizing Exported Function-Call Subsystems

You can use Real-Time Workshop options to optimize the code generated for a function-call subsystem or virtual block that contains such subsystems. To obtain faster code,

- Specify a storage class for every input signal and output signal that crosses the boundary of the subsystem.
- For each function-call subsystem to be exported (whether directly or within a virtual subsystem):
  - **a** Right-click the subsystem and choose **Subsystem Parameters** from the context menu.
  - **b** Set the **Real-Time Workshop system code** parameter to `Auto`.
  - **c** Click **OK** or **Apply**.

Exporting Function-Call Subsystems That Depend on Elapsed Time

Some blocks, such as the Sine Wave block (if sample-based) and the Discrete-Time Integrator block, depend on elapsed time. See “Absolute and

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

When a block that depends on elapsed time exists in a function-call subsystem, the subsystem cannot be exported unless it specifies periodic execution. To provide the necessary specification,

1 Right-click the trigger port block in the function-call subsystem and choose **TriggerPort Parameters** from the context menu.

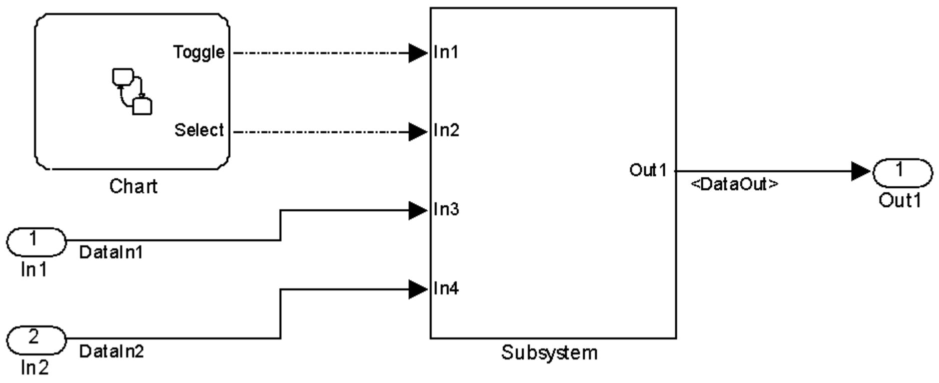
2 Specify **periodic** in the **Sample time type** field.

3 Set the **Sample time** to the same granularity specified (directly or by inheritance) in the function-call initiator.

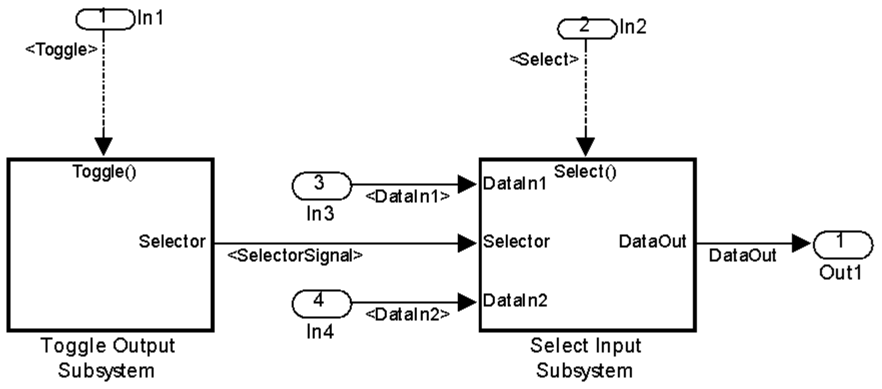
4 Click **OK** or **Apply**.

Function-Call Subsystem Export Example

The next figure shows the top level of a model that uses a Stateflow chart named `Chart` to input two function-call trigger signals (denoted by dash-dot lines) to a virtual subsystem named `Subsystem`.



The next figure shows the contents of `Subsystem` in the previous figure. The subsystem contains two function-call subsystems, each driven by one of the signals input from the top level.



In the preceding model, the Stateflow chart can assert either of two scalar signals, `Toggle` and `Select`.

- Asserting `Toggle` toggles the Boolean state of the function-call subsystem `Toggle Output Subsystem`.
- Asserting `Select` causes the function-call subsystem `Select Input Subsystem` to assign the value of `DataIn1` or `DataIn2` to its output signal. The value assigned depends on the current state of `Toggle Output Subsystem`.

The following generated code implements the subsystem named `Subsystem`. The code is typical for virtual subsystems that contain function-call subsystems. It specifies an initialization function and a function for each contained subsystem, and would also include functions to enable and disable subsystems if applicable.

```
#include "Subsystem.h"
#include "Subsystem_private.h"

/* Exported block signals */
real_T DataIn1;                        /* '<Root>/In3' */
real_T DataIn2;                        /* '<Root>/In4' */
real_T DataOut;                        /* '<S4>/Switch' */
boolean_T SelectorSignal;              /* '<S5>/Logical Operator' */

/* Exported block states */
```

```
boolean_T SelectorState;                /* '<S5>/Unit Delay' */

/* Real-time model */
RT_MODEL_Subsystem Subsystem_M_;
RT_MODEL_Subsystem *Subsystem_M = &Subsystem_M_;

/* Initial conditions for exported function: Toggle */

void Toggle_Init(void)
{
  /* Initial conditions for function-call system: '<S1>/Toggle Output Subsystem' */

  /* InitializeConditions for UnitDelay: '<S5>/Unit Delay' */
  SelectorState = Subsystem_P.UnitDelay_X0;
}

/* Output and update for exported function: Toggle */

void Toggle(void)
{
  /* Output and update for function-call system: '<S1>/Toggle Output Subsystem' */

  /* Logic: '<S5>/Logical Operator' incorporates:
   *  UnitDelay: '<S5>/Unit Delay'
   */
  SelectorSignal = !SelectorState;

  /* Update for UnitDelay: '<S5>/Unit Delay' */
  SelectorState = SelectorSignal;
}

/* Output and update for exported function: Select */

void Select(void)
{
  /* Output and update for function-call system: '<S1>/Select Input Subsystem' */

  /* Switch: '<S4>/Switch' incorporates:
   *  Inport: '<Root>/In3'
   *  Inport: '<Root>/In4'
   */
  if(SelectorSignal) {
    DataOut = DataIn1;
  } else {
    DataOut = DataIn2;
  }
}

/* Model initialize function */

void Subsystem_initialize(void)
{
  /* initialize error status */
  rtmSetErrorStatus(Subsystem_M, (const char_T *)0);

  /* block I/O */

  /* exported global signals */
  DataOut = 0.0;
  SelectorSignal = FALSE;

  /* states (dwork) */

  /* exported global states */
  SelectorState = FALSE;

  /* external inputs */
  DataIn1 = 0.0;
  DataIn2 = 0.0;

  Toggle_Init();
}

/* Model terminate function */

void Subsystem_terminate(void)
{
  /* (no terminate code required) */
}
```

Function-Call Subsystems Export Limitations

The function-call subsystem export capabilities have the following limitations:

- Real-Time Workshop options do not control the names of the files containing the generated code. All such filenames begin with the name of the exported subsystem. Each filename is suffixed as appropriate to the file.
- Real-Time Workshop options do not control the names of top-level functions in the generated code. Each function name reflects the name of the signal that triggers the function, or for an unnamed signal, the block from which the signal originates.
- This release cannot export reusable code for a function-call subsystem. Checking **Configuration Parameters > Real-Time Workshop > Interface > Generate reusable code** has no effect on the generated code for the subsystem.
- This release supports code generation for ERT generated S-function blocks if the block does not have function-call input ports, but the ERT S-function block will appear as a noninlined S-function in the generated code.
- This release supports an ERT generated S-function block in accelerator mode only if its function-call initiator is noninlined in accelerator mode. Examples of noninlined initiators include all Stateflow charts.
- The ERT S-function wrapper must be driven by a Level-2 S-function initiator block, such as a Stateflow chart or the built-in Function-call Generator block.
- An asynchronous (sample-time) function-call system can be exported, but this release does not support the ERT S-function wrapper for an asynchronous system.
- This release does not support code generation for an ERT generated S-function block if the block was generated as a wrapper for exported function calls.
- The output port of an ERT generated S-function block cannot be merged using the Merge block.
- This release does not support MAT-file logging for exported function calls. Any specification that enables MAT-file logging is ignored.
- The use of the TLC function `LibIsFirstInit` is deprecated for exported function calls.

- The *model*`_initialize` function generated in the code for an exported function-call subsystem never includes a `firstTime` argument, regardless of the value of the model configuration parameter `IncludeERTFirstTime`. Thus, you cannot call *model*`_initialize` at a time greater than start time, for example, to reset block states.

Nonvirtual Subsystem Modular Function Code Generation

In this section...

Overview

Real-Time Workshop Embedded Coder provides a subsystem option, **Function with separate data**, that allows you to generate modular function code for nonvirtual subsystems, including atomic subsystems and conditionally executed subsystems.

By default, the generated code for a nonvirtual subsystem does not separate a subsystem's internal data from the data of its parent Simulink model. This can make it difficult to trace and test the code, particularly for nonreusable subsystems. Also, in large models containing nonvirtual subsystems, data structures can become large and potentially difficult to compile.

The Subsystem Parameters dialog box option **Function with separate data** allows you to generate subsystem function code in which the internal data for a nonvirtual subsystem is separated from its parent model and is owned by the subsystem. As a result, the generated code for the subsystem is easier to trace and test. The data separation also tends to reduce the size of data structures throughout the model.

Note Selecting the **Function with separate data** option for a nonvirtual subsystem has no semantic effect on the parent Simulink model.

To be able to use this option,

- Your Simulink model must use an ERT-based system target file (requires a license for Real-Time Workshop Embedded Coder).
- Your subsystem must be configured to be atomic or conditionally executed (for more information, see “Systems and Subsystems” in the Simulink documentation).
- Your subsystem must use the `Function` setting for the **Real-Time Workshop system code** parameter.

To configure your subsystem for generating modular function code, you invoke the Subsystem Parameters dialog box and make a series of selections to display and enable the **Function with separate data** option. See “Configuring Nonvirtual Subsystems for Generating Modular Function Code” on page 2-76 and “Examples of Modular Function Code for Nonvirtual Subsystems” on page 2-80 for details. For limitations that apply, see “Nonvirtual Subsystem Modular Function Code Limitations” on page 2-86.

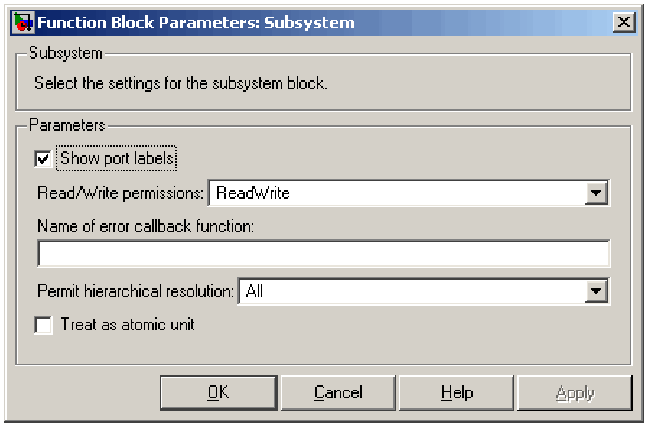
For more information about generating code for atomic subsystems, see the sections “Nonvirtual Subsystem Code Generation” and “Generating Code and Executables from Subsystems” in the Real-Time Workshop documentation.

Configuring Nonvirtual Subsystems for Generating Modular Function Code

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

1. Verify that the Simulink model containing the subsystem uses an ERT-based system target file (see the **System target file** parameter on the **Real-Time Workshop** pane of the Configuration Parameters dialog box).

2. In your Simulink model, select the subsystem for which you want to generate modular function code and launch the Subsystem Parameters dialog box (for example, right-click the subsystem and select **Subsystem Parameters**). The dialog box for an atomic subsystem is shown below. (In the dialog box for a conditionally executed subsystem, the dialog box option **Treat as atomic unit** is greyed out, and you can skip Step 3.)


Function Block Parameters: Subsystem

Subsystem

Select the settings for the subsystem block.

Parameters

☑ Show port labels

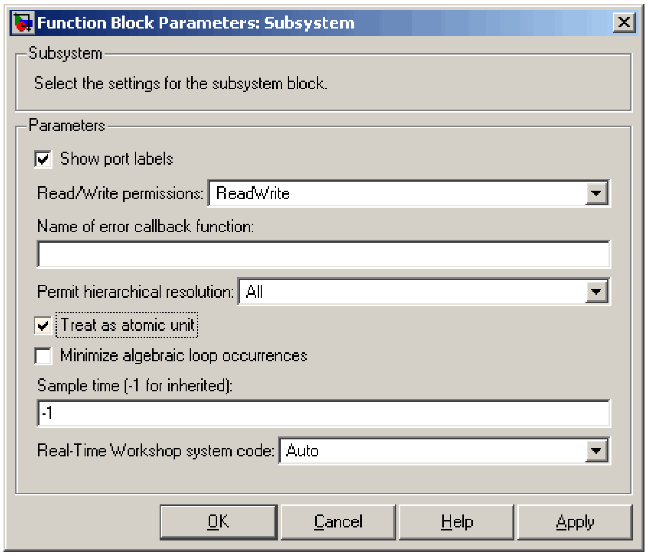
Read/Write permissions: ReadWrite

Name of error callback function:

Permit hierarchical resolution: All

☐ Treat as atomic unit

3 If the Subsystem Parameters dialog box option **Treat as atomic unit** is available for selection but not selected, the subsystem is neither atomic nor conditionally executed. Select the option **Treat as atomic unit**. After you make this selection, the **Real-Time Workshop system code** parameter is displayed.


Function Block Parameters: Subsystem

Subsystem

Select the settings for the subsystem block.

Parameters

☑ Show port labels

Read/Write permissions: ReadWrite

Name of error callback function:

Permit hierarchical resolution: All

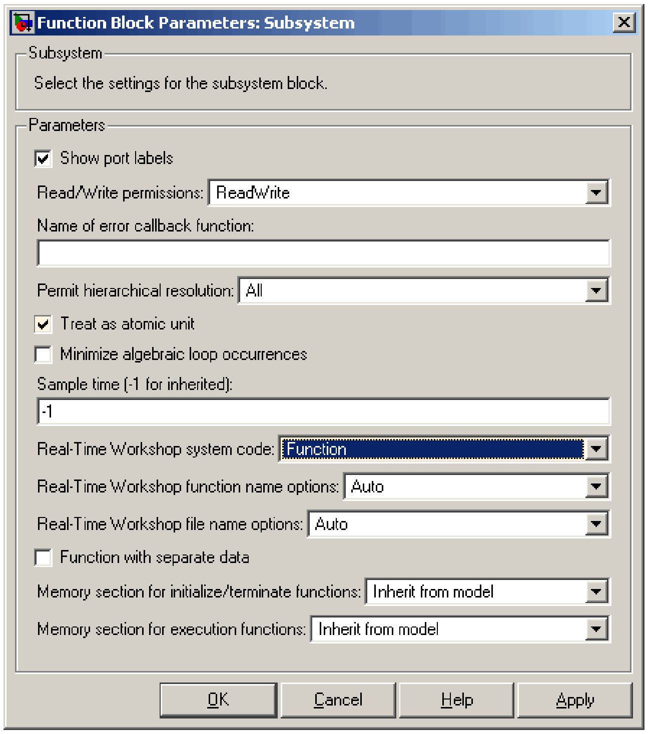
☑ Treat as atomic unit

☐ Minimize algebraic loop occurrences

Sample time (-1 for inherited): -1

Real-Time Workshop system code: Auto

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



Function Block Parameters: Subsystem

Subsystem

Select the settings for the subsystem block.

Parameters

[✓] Show port labels

Read/Write permissions: ReadWrite

Name of error callback function:

Permit hierarchical resolution: All

[✓] Treat as atomic unit

[ ] Minimize algebraic loop occurrences

Sample time (-1 for inherited): -1

Real-Time Workshop system code: Function

Real-Time Workshop function name options: Auto

Real-Time Workshop file name options: Auto

[ ] Function with separate data

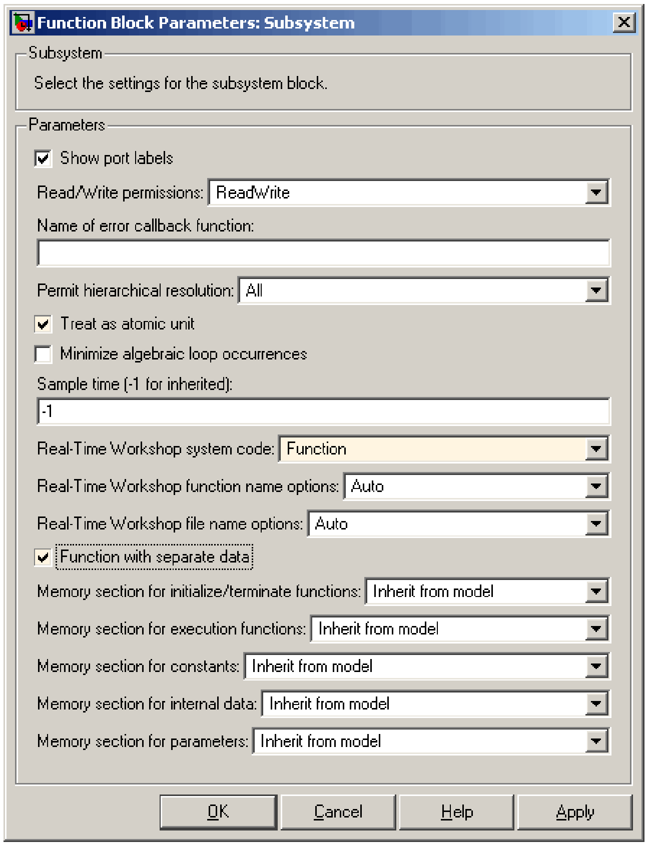
Memory section for initialize/terminate functions: Inherit from model

Memory section for execution functions: Inherit from model

Note Before you generate nonvirtual subsystem function code with the **Function with separate data** option selected, you might want to generate function code with the option *deselected* and save the generated function `.c` and `.h` files in a separate directory for later comparison.

---

5 Select the **Function with separate data** option. After you make this selection, additional configuration parameters are displayed.


Function Block Parameters: Subsystem

Subsystem

Select the settings for the subsystem block.

Parameters

[✓] Show port labels

Read/Write permissions: ReadWrite

Name of error callback function:

Permit hierarchical resolution: All

[✓] Treat as atomic unit

[ ] Minimize algebraic loop occurrences

Sample time (-1 for inherited): -1

Real-Time Workshop system code: Function

Real-Time Workshop function name options: Auto

Real-Time Workshop file name options: Auto

[✓] Function with separate data

Memory section for initialize/terminate functions: Inherit from model

Memory section for execution functions: Inherit from model

Memory section for constants: Inherit from model

Memory section for internal data: Inherit from model

Memory section for parameters: Inherit from model

Note To control the naming of the subsystem function and the subsystem files in the generated code, you can modify the subsystem parameters **Real-Time Workshop function name options** and **Real-Time Workshop file name options**.

---

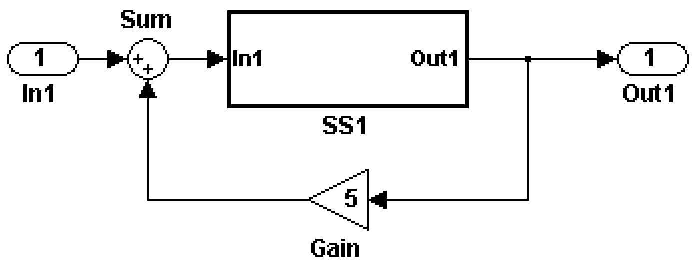
6 To save your subsystem parameter settings and exit the dialog box, click **OK**.

This completes the subsystem configuration needed to generate modular function code. You can now generate the code for the subsystem and examine the generated files, including the function `.c` and `.h` files named according to your subsystem parameter specifications. For more information on generating code for nonvirtual subsystems, see “Nonvirtual Subsystem Code Generation” in the Real-Time Workshop documentation. For examples of generated subsystem function code, see “Examples of Modular Function Code for Nonvirtual Subsystems” on page 2-80.

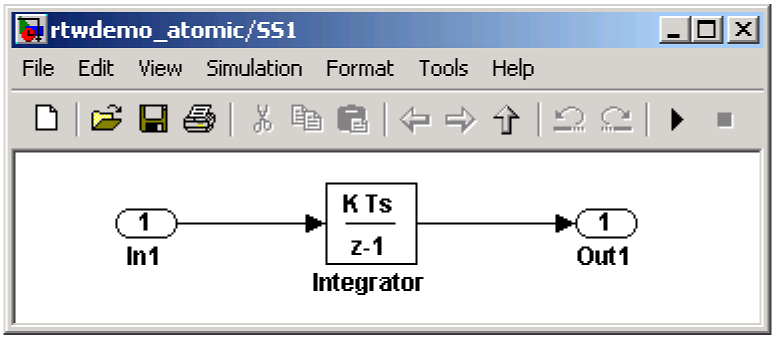
Examples of Modular Function Code for Nonvirtual Subsystems

To illustrate the effect of selecting the **Function with separate data** option for a nonvirtual subsystem, the following procedure generates atomic subsystem function code with and without the option selected and compares the results.

1 Open MATLAB and launch `rtwdemo_atomic.mdl` using the MATLAB command `rtwdemo_atomic`. Examine the Simulink model.



2 Double-click the SS1 subsystem and examine the contents. (You can close the subsystem window when you are finished.)



3 Use the Configuration Parameters dialog box to change the model's **System target file** from GRT to ERT. For example, from the Simulink window, select **Simulation > Configuration Parameters**, select the **Real-Time Workshop** pane, select **System target file** `ert.tlc`, and click **OK** twice to confirm the change.

4 Create a variant of `rtwdemo_atomic.mdl` that illustrates function code *without* data separation.

a In the Simulink view of `rtwdemo_atomic.mdl`, right-click the SS1 subsystem and select **Subsystem Parameters**. In the Subsystem Parameters dialog box, verify that

- **Treat as atomic unit** is checked
- `User specified` is selected as the value for the **Real-Time Workshop function name options** parameter
- `myfun` is specified as the value for the **Real-Time Workshop function name** parameter

b In the Subsystem Parameters dialog box,

i Select the value `Function` for the **Real-Time Workshop system code** parameter. After this selection, additional parameters and options will appear.

ii Select the value `Use function name` for the **Real-Time Workshop file name** parameter. This selection is optional but simplifies the later task of code comparison by causing the atomic subsystem function code to be generated into the files `myfun.c` and `myfun.h`.

Do *not* select the option **Function with separate data**. Click **Apply** to apply the changes and click **OK** to exit the dialog box.

c Save this model variant to a personal work directory, for example, `d:/atomic/rtwdemo_atomic1.mdl`.

5 Create a variant of `rtwdemo_atomic.mdl` that illustrates function code *with* data separation.

a In the Simulink view of `rtwdemo_atomic1.mdl` (or `rtwdemo_atomic.mdl` with step 3 reapplied), right-click the SS1 subsystem and select **Subsystem Parameters**. In the Subsystem Parameters dialog box, verify that

- **Treat as atomic unit** is checked
- `Function` is selected for the **Real-Time Workshop system code** parameter
- `User specified` is selected as the value for the **Real-Time Workshop function name options** parameter
- `myfun` is specified as the value for the **Real-Time Workshop function name** parameter
- `Use function name` is selected for the **Real-Time Workshop file name options** parameter

b In the Subsystem Parameters dialog box, select the option **Function with separate data**. Click **Apply** to apply the change and click **OK** to exit the dialog box.

c Save this model variant, using a different name than the first variant, to a personal work directory, for example, `d:/atomic/rtwdemo_atomic2.mdl`.

6 Generate code for each model, `d:/atomic/rtwdemo_atomic1.mdl` and `d:/atomic/rtwdemo_atomic2.mdl`.

7 In the generated code directories, compare the *`model`*`.c`/`.h` and `myfun.c`/`.h` files generated for the two models. (In this example, there are no significant differences in the generated variants of `ert_main.c`, *`model`*`_private.h`, *`model`*`_types.h`, or `rtwtypes.h`.)

H File Differences for Nonvirtual Subsystem Function Data Separation

The differences between the H files generated for `rtwdemo_atomic1.mdl` and `rtwdemo_atomic2.mdl` help illustrate the effect of selecting the **Function with separate data** option for nonvirtual subsystems.

1. Selecting **Function with separate data** causes typedefs for subsystem data to be generated in the `myfun.h` file for `rtwdemo_atomic2`:

   ```
   /* Block signals for system '<Root>/SS1' */
   typedef struct {
     real_T Integrator;                    /* '<S1>/Integrator' */
   } rtB_myfun;

   /* Block states (auto storage) for system '<Root>/SS1' */
   typedef struct {
     real_T Integrator_DSTATE;             /* '<S1>/Integrator' */
   } rtDW_myfun;
   ```

   By contrast, for `rtwdemo_atomic1`, typedefs for subsystem data belong to the model and appear in `rtwdemo_atomic1.h`:

   ```
   /* Block signals (auto storage) */
   typedef struct {
   ...
       real_T Integrator;                  /* '<S1>/Integrator' */
   } BlockIO_rtwdemo_atomic1;

   /* Block states (auto storage) for system '<Root>' */
   typedef struct {
     real_T Integrator_DSTATE;             /* '<S1>/Integrator' */
   } D_Work_rtwdemo_atomic1;
   ```

2. Selecting **Function with separate data** generates the following external declarations in the `myfun.h` file for `rtwdemo_atomic2`:

   ```
   /* Extern declarations of internal data for 'system '<Root>/SS1'' */
   extern rtB_myfun rtwdemo_atomic2_myfunB;

   extern rtDW_myfun rtwdemo_atomic2_myfunDW;
   ```

```
extern void myfun_initialize(void);
```

By contrast, the generated code for `rtwdemo_atomic1` contains model-level external declarations for the subsystem's `BlockIO` and `D_Work` data, in `rtwdemo_atomic1.h`:

```
/* Block signals (auto storage) */
extern BlockIO_rtwdemo_atomic1 rtwdemo_atomic1_B;

/* Block states (auto storage) */
extern D_Work_rtwdemo_atomic1 rtwdemo_atomic1_DWork;
```

C File Differences for Nonvirtual Subsystem Function Data Separation

The differences between the C files generated for `rtwdemo_atomic1.mdl` and `rtwdemo_atomic2.mdl` illustrate the key effects of selecting the **Function with separate data** option for nonvirtual subsystems.

1 Selecting **Function with separate data** causes a separate subsystem initialize function, `myfun_initialize`, to be generated in the `myfun.c` file for `rtwdemo_atomic2`:

```
void myfun_initialize(void) {
  {
    ((real_T*)&rtwdemo_atomic2_myfunB.Integrator)[0] = 0.0;
  }
  rtwdemo_atomic2_myfunDW.Integrator_DSTATE = 0.0;
}
```

The subsystem initialize function in `myfun.c` is invoked by the model initialize function in `rtwdemo_atomic2.c`:

```
/* Model initialize function */

void rtwdemo_atomic2_initialize(void)
{
...

  /* Initialize subsystem data */
  myfun_initialize();
}
```

By contrast, for `rtwdemo_atomic1`, subsystem data is initialized by the model initialize function in `rtwdemo_atomic1.c`:

```
/* Model initialize function */

void rtwdemo_atomic1_initialize(void)
{
...
  /* block I/O */
  {
 ...
    ((real_T*)&rtwdemo_atomic1_B.Integrator)[0] = 0.0;
  }

  /* states (dwork) */

  rtwdemo_atomic1_DWork.Integrator_DSTATE = 0.0;
...
}
```

2 Selecting **Function with separate data** generates the following declarations in the `myfun.c` file for `rtwdemo_atomic2`:

```
/* Declare variables for internal data of system '<Root>/SS1' */
rtB_myfun rtwdemo_atomic2_myfunB;

rtDW_myfun rtwdemo_atomic2_myfunDW;
```

By contrast, the generated code for `rtwdemo_atomic1` contains model-level declarations for the subsystem's `BlockIO` and `D_Work` data, in `rtwdemo_atomic1.c`:

```
/* Block signals (auto storage) */
BlockIO_rtwdemo_atomic1 rtwdemo_atomic1_B;

/* Block states (auto storage) */
D_Work_rtwdemo_atomic1 rtwdemo_atomic1_DWork;
```

3 Selecting **Function with separate data** generates identifier naming that reflects the subsystem orientation of data items. Notice the references to subsystem data in subsystem functions such as `myfun` and `myfun_update` or in the model's *`model`*`_step` function. For example, compare this code from `myfun` for `rtwdemo_atomic2`

```
/* DiscreteIntegrator: '<S1>/Integrator' */
rtwdemo_atomic2_myfunB.Integrator = rtwdemo_atomic2_myfunDW.Integrator_DSTATE;
```

to the corresponding code from `myfun` for `rtwdemo_atomic1`.

```
/* DiscreteIntegrator: '<S1>/Integrator' */
rtwdemo_atomic1_B.Integrator = rtwdemo_atomic1_DWork.Integrator_DSTATE;
```

Nonvirtual Subsystem Modular Function Code Limitations

The nonvirtual subsystem option **Function with separate data** has the following limitations:

- The **Function with separate data** option is available only in ERT-based Simulink models (requires a license for Real-Time Workshop Embedded Coder).
- The nonvirtual subsystem to which the option is applied cannot have multiple sample times or continuous sample times; that is, the subsystem must be single-rate with a discrete sample time.
- The nonvirtual subsystem cannot contain continuous states.
- The nonvirtual subsystem cannot output function call signals.
- The nonvirtual subsystem cannot contain non-inlined S-functions.
- The generated files for the nonvirtual subsystem will reference model-wide header files, such as *`model`*`.h` and *`model`*`_private.h`.
- The **Function with separate data** option is incompatible with the **GRT compatible call interface** option, located on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box. Selecting both will generate an error.

- The **Function with separate data** option is incompatible with the **Generate reusable code** option (**Real-Time Workshop/Interface** pane). Selecting both will generate an error.
- Although the *model*`_initialize` function generated for a model containing a nonvirtual subsystem that uses the **Function with separate data** option may have a `firstTime` argument, the argument is not used. Thus, you cannot call *model*`_initialize` at a time greater than start time, for example, to reset block states. To suppress inclusion of the `firstTime` flag in the *model*`_initialize` function definition, set the model configuration parameter `IncludeERTFirstTime` to `off`.

Controlling model_step Function Prototypes

In this section...

Overview

Real-Time Workshop Embedded Coder provides a **Configure Functions** button, located on the **Interface** pane of the Configuration Parameters dialog box, that allows you to control the `model_step` function prototype that is generated for ERT-based Simulink models.

By default, the function prototype of an ERT-based model's generated `model_step` function resembles the following:

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

If you generate reusable, reentrant code for an ERT-based model, the model's root-level inputs and outputs, block states, parameters, and external outputs are passed in to `model_step` using a function prototype that resembles the following:

```
void model_step(inport_args, outport_args, BlockIO_arg, DWork_arg, RT_model_arg);
```

(For more detailed information about the default calling interface for the `model_step` function, see the `model_step` reference page.)

The **Configure Functions** button on the **Interface** pane provides you flexible control over the *model*_step function prototype that is generated for your model. Clicking **Configure Functions** launches a Model Step Functions dialog box (see "Model Step Functions Dialog Box" on page 2-89). Based on the **Function specification** value you specify for your *model*_step function (supported values include `Default model-step function` and `Model specific C prototype`), you can preview and modify the function prototype. Once you validate and apply your changes, you can generate code based on your function prototype modifications.

For more information about using the **Configure Functions** button and the Model Step Functions dialog box, see "model_step Function Prototype Example" on page 2-92. See also the demo model `rtwdemo_fcnprotoctrl`, which is preconfigured to demonstrate function prototype control.

Alternatively, you can use function prototype control functions to programmatically control *model*_step function prototypes. For more information, see "Configuring a model_step Function Prototype Programmatically" on page 2-97.

You can also control step function prototypes for nonvirtual subsystems, if you generate subsystem code using right-click build. To launch the Model Step Functions for Subsystem dialog box, use the function `RTW.configSubsystemBuild`:

```
RTW.configSubsystemBuild('model/subsystem')
RTW.configSubsystemBuild(gcb)
```

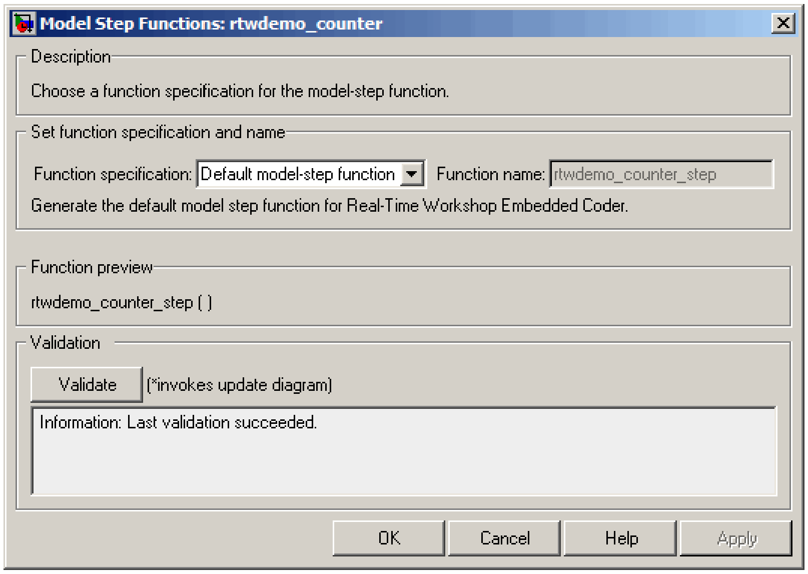
Right-click building the subsystem will generate the step function according to the customizations you make. For more information, see "Configuring a Step Function Prototype for a Nonvirtual Subsystem" on page 2-101.

For limitations that apply, see "model_step Function Prototype Control Limitations" on page 2-103.

Model Step Functions Dialog Box

Clicking the **Configure Functions** button on the **Interface** pane launches the Model Step Functions dialog box. This dialog box is the starting point for configuring the *model*_step function prototype that is generated during code generation for ERT-based Simulink models. Based on the **Function specification** value you select for your `*model*_step` function (supported values include `Default model-step function` and `Model specific C prototype`), you can preview and modify the function prototype. Once you validate and apply your changes, you can generate code based on your function prototype modifications.

The figure below shows the Model Step Functions dialog box in the `Default model-step function` view.



Model Step Functions: rtwdemo_counter

Description

Choose a function specification for the model-step function.

Set function specification and name

Function specification: Default model-step function   Function name: rtwdemo_counter_step

Generate the default model step function for Real-Time Workshop Embedded Coder.

Function preview

rtwdemo_counter_step ( )

Validation

Validate (*invokes update diagram)

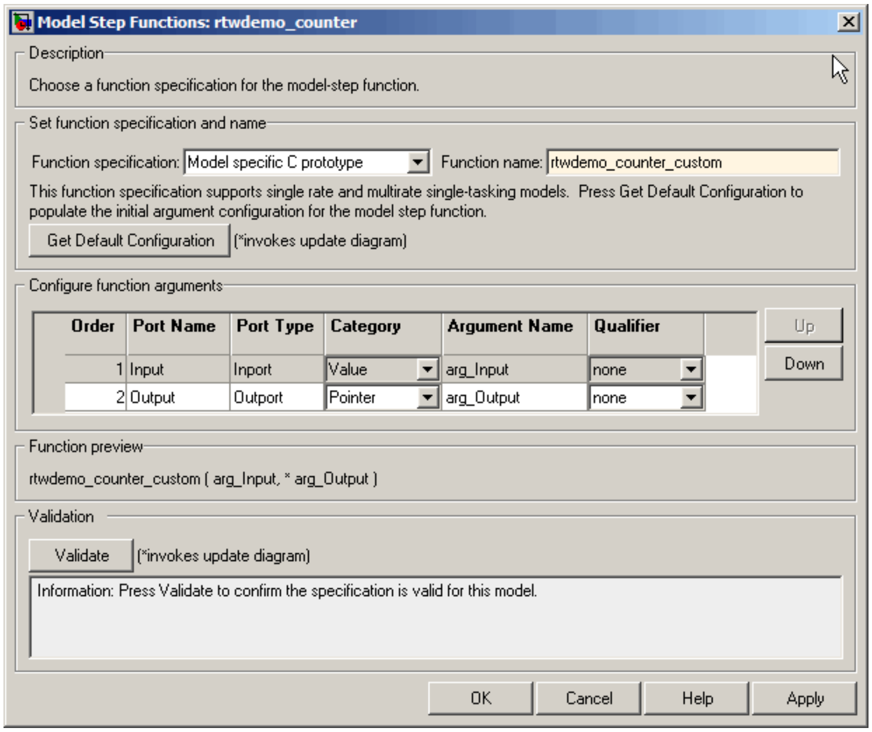
Information: Last validation succeeded.

The `Default model-step function` view allows you to validate and preview the predicted default `*model*_step` function prototype. To validate the default function prototype configuration against your model, click the **Validate** button. If the validation succeeds, the predicted function prototype will be displayed in the **Function preview** subpane.

Note You can not use the `Default model-step function` view to modify the function prototype configuration.

Selecting `Model specific C prototype` for the **Function specification** parameter displays the `Model specific C prototype` view of your *`model`*`_step` function. This view provides controls that you can use to customize the function name, the order of arguments, and argument attributes including name, passing mechanism, and type qualifier for each of the model's root-level I/O ports.

To begin configuring your function control prototype configuration, click the **Get Default Configuration** button. This activates the **Configure function arguments** subpane, as shown below.


Model Step Functions: rtwdemo_counter

Description

Choose a function specification for the model-step function.

Set function specification and name

Function specification: Model specific C prototype   Function name: rtwdemo_counter_custom

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model step function.

Get Default Configuration (*invokes update diagram)

Configure function arguments

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	Input	Inport	Value	arg_Input	none
2	Output	Outport	Pointer	arg_Output	none

Function preview

rtwdemo_counter_custom (arg_Input, * arg_Output)

Validation

Validate (*invokes update diagram)

Information: Press Validate to confirm the specification is valid for this model.

In the **Configure function arguments** subpane:

- **Category** specifies how an argument is passed in or out from the customized step function, either by copying a value (`Value`) or by a pointer to a memory space (`Pointer`).

- **Qualifier** (optional), specifies a `const` type qualifier for a function argument. The possible values are `none`, `const` (value), `const*` (value referenced by the pointer), and `const*const` (value referenced by the pointer and the pointer itself).

The **Function preview** subpane provides a preview of how your function prototype will be interpreted in generated code. The preview is updated dynamically as you make modifications.

An argument `foo` whose **Category** is `Pointer` will be previewed as `* foo`. If its **Category** is `Value`, it will be previewed as `foo`. Notice that argument types and qualifiers are not represented in the **Function preview** subpane.

model_step Function Prototype Example

The following procedure demonstrates how to use the **Configure Functions** button on the **Interface** pane of the Configuration Parameters dialog box to control the *`model`*`_step` function prototype that is generated for your Simulink model.

1 Open MATLAB and launch `rtwdemo_counter.mdl` using the MATLAB command `rtwdemo_counter`.

2 In the `rtwdemo_counter` display, double-click the button **Generate Code Using Real-Time Workshop Embedded Coder (double-click)**. This button generates code for an ERT-based version of `rtwdemo_counter.mdl`. The code generation report for `rtwdemo_counter` is displayed.

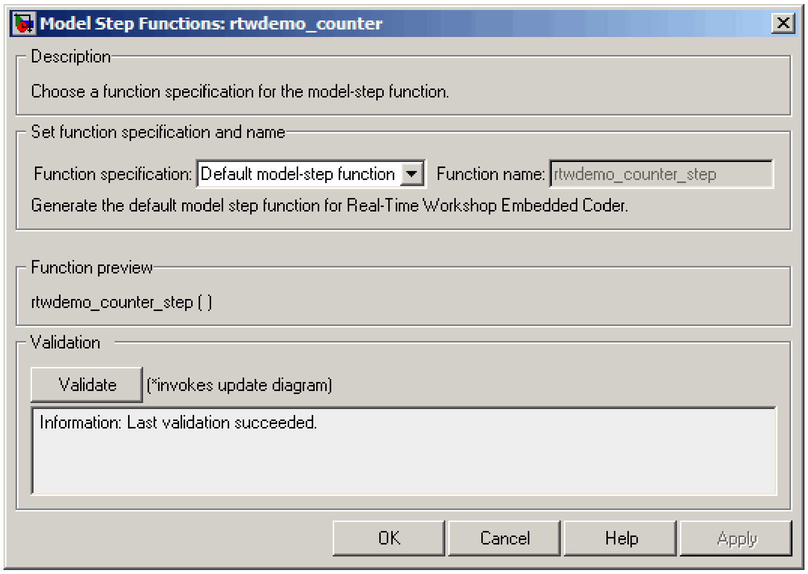
3 In the code generation report, click the link for `rtwdemo_counter.c`. In the `rtwdemo_counter.c` code display, locate and examine the generated code for the function `rtwdemo_counter_step`, beginning with

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

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

4 From the Simulink window for `rtwdemo_counter`, launch the Configuration Parameters dialog box. Go to the **Interface** pane and launch the Model Step Functions dialog box by clicking the **Configure Functions** button.

5 In the initial (`Default model-step funtion`) view of the Model Step Functions dialog box, click the **Validate** button to validate and preview the default function prototype for the `rtwdemo_counter_step` function. The function prototype arguments displayed under **Function preview** should correspond to the default prototype generated in step 3.


Model Step Functions: rtwdemo_counter

Description

Choose a function specification for the model-step function.

Set function specification and name

Function specification: Default model-step function   Function name: rtwdemo_counter_step

Generate the default model step function for Real-Time Workshop Embedded Coder.

Function preview

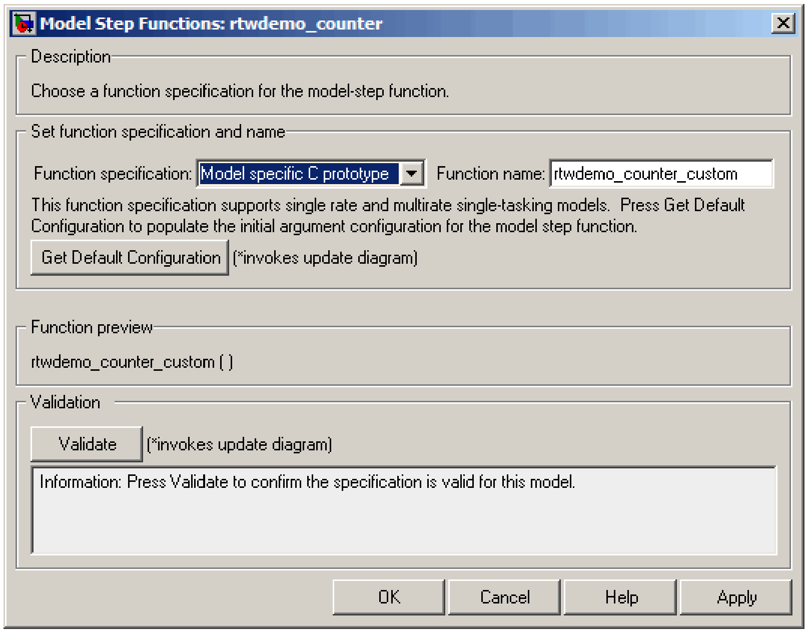
rtwdemo_counter_step ( )

Validation

Validate (*invokes update diagram)

Information: Last validation succeeded.

6 In the Model Step Functions dialog box, set **Function specification** to `Model specific C prototype`. Making this selection switches the dialog box from the `Default model-step function` view to the `Model specific C prototype` view.



Model Step Functions: rtwdemo_counter

Description

Choose a function specification for the model-step function.

Set function specification and name

Function specification: Model specific C prototype Function name: rtwdemo_counter_custom

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model step function.

Get Default Configuration (*invokes update diagram)

Function preview

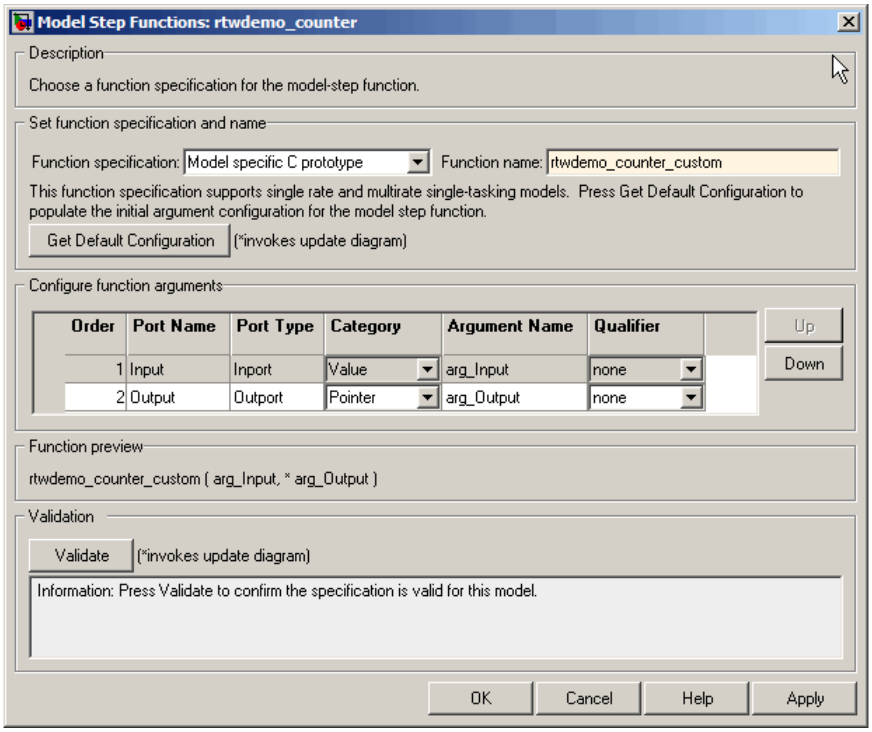
rtwdemo_counter_custom ( )

Validation

Validate (*invokes update diagram)

Information: Press Validate to confirm the specification is valid for this model.

7 In the `Model specific C prototype` view, click the **Get Default Configuration** button to activate the **Configure function arguments** subpane.


Model Step Functions: rtwdemo_counter
Description
Set function specification and name
Get Default Configuration
Configure function arguments
Up
Down
Function preview
Validation
Validate
OK
Cancel
Help
Apply

Choose a function specification for the model-step function.

Function specification: Model specific C prototype

Function name: rtwdemo_counter_custom

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model step function.

(*invokes update diagram)

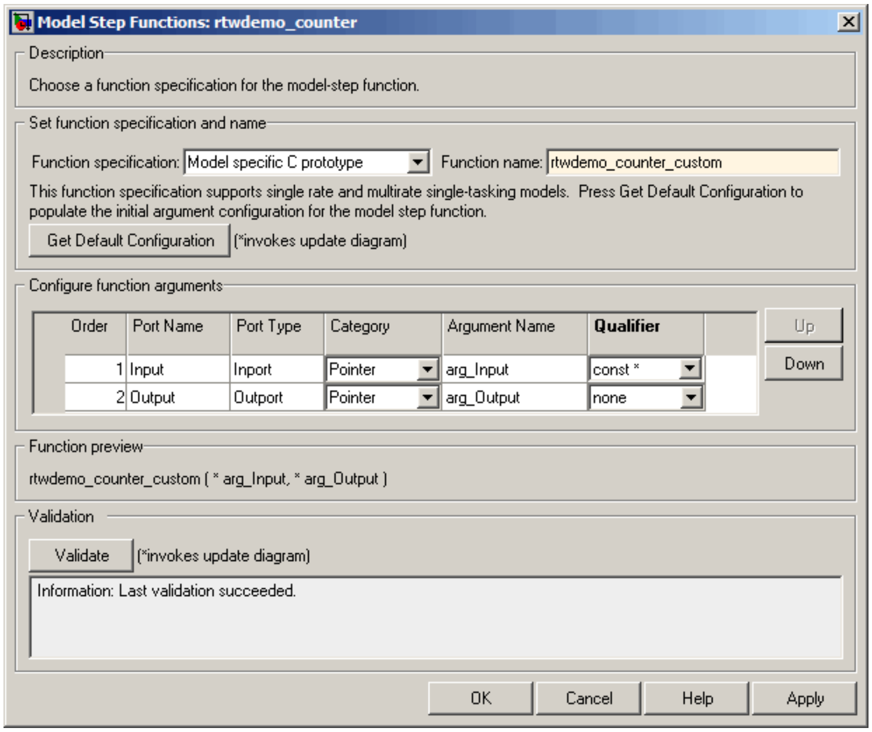
Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	Input	Inport	Value	arg_Input	none
2	Output	Outport	Pointer	arg_Output	none

rtwdemo_counter_custom (arg_Input, * arg_Output)

(*invokes update diagram)

Information: Press Validate to confirm the specification is valid for this model.

8 In the **Configure function arguments** subpane, in the row for the `Input` argument, change the value of **Category** from `Value` to `Pointer` and change the value of **Qualifier** from `none` to `const *`. The preview is updated to reflect your changes. Click the **Validate** button to validate the modified function prototype.


Model Step Functions: rtwdemo_counter
Description
Set function specification and name
Get Default Configuration
Configure function arguments
Up
Down
Function preview
Validation
Validate
OK
Cancel
Help
Apply

Choose a function specification for the model-step function.

Function specification: Model specific C prototype   Function name: rtwdemo_counter_custom

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model step function.

(*invokes update diagram)

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	Input	Inport	Pointer	arg_Input	const *
2	Output	Outport	Pointer	arg_Output	none

rtwdemo_counter_custom (* arg_Input, * arg_Output)

(*invokes update diagram)

Information: Last validation succeeded.

9 Click **OK** to exit the Model Step Functions dialog box and then generate code for the model (for example, by using Ctrl/B or the Real-Time Workshop **Build** button). When the build completes, the code generation report for `rtwdemo_counter` will be displayed.

10 In the code generation report, click the link for `rtwdemo_counter.c`. In the `rtwdemo_counter.c` code display, locate and examine the generated code for the function `rtwdemo_counter_custom`, beginning with

```
/* Customized model step function */
void rtwdemo_counter_custom(const int32_T *arg_Input, int32_T *arg_Output)
{
 ...
}
```

Verify that the generated code is consistent with the function prototype modifications that you specified using the Model Step Functions dialog box.

Configuring a model_step Function Prototype Programmatically

You can use the function prototype control functions (listed below in Function Prototype Control Functions on page 2-98), to programmatically control `model_step` function prototypes. Typical uses of the listed functions include:

- **Create and validate a new function prototype**

  a Create a model-specific C function prototype with `obj = RTW.ModelSpecificCPrototype`, where *`obj`* returns a handle to an newly-created, empty function prototype.

  b Add argument configuration information for your model ports using `addArgConf`.

  c Attach the function prototype to your loaded ERT-based Simulink model using `attachToModel`.

  d Validate the function prototype using `runValidation`.

  e If validation succeeds, save your model and then generate code using `rtwbuild`.

- **Modify and validate an existing function prototype**

  a Get the handle to an existing model-specific C function prototype that is attached to your loaded ERT-based Simulink model with `obj = RTW.getFunctionSpecification(modelName)`, where *`modelName`* is a string specifying the name of a loaded ERT-based Simulink model, and *`obj`* returns a handle to a function prototype attached to the specified model.

  You can use other function prototype control functions on the returned handle only if the test `isa(obj,'RTW.ModelSpecificCPrototype')` returns 1. If the model does not have a function prototype configuration, the function returns `[]`. If the function returns a handle to an object of type `RTW.FcnDefault`, you cannot modify the existing function prototype.

  b Use the `Get` and `Set` functions listed in Function Prototype Control Functions on page 2-98 to test and/or reset such items as the function name, argument names, argument positions, argument categories, and argument type qualifiers.

  c Validate the function prototype using `runValidation`.

d If validation succeeds, save your model and then generate code using `rtwbuild`.

- **Create and validate a new function prototype, starting with default configuration information from your Simulink model**

  a Create a model-specific C function prototype using `*obj* = RTW.ModelSpecificCPrototype`, where *`obj`* returns a handle to an newly-created, empty function prototype.

  b Attach the function prototype to your loaded ERT-based Simulink model using `attachToModel`.

  c Get default configuration information from your model using `getDefaultConf`.

  d Use the `Get` and `Set` functions listed in Function Prototype Control Functions on page 2-98 to test and/or reset such items as the function name, argument names, argument positions, argument categories, and argument type qualifiers.

  e Validate the function prototype using `runValidation`.

  f If validation succeeds, save your model and then generate code using `rtwbuild`.

Note You should not use the same model-specific C function prototype object across multiple models. If you do, changes that you make to the step function prototype configuration in one model will be propagated to other models, which is usually not desirable.

Function Prototype Control Functions

Function	Description
`addArgConf`	Add argument configuration information for Simulink model port to model-specific C function prototype
`attachToModel`	Attach model-specific C function prototype to loaded ERT-based Simulink model
`getArgCategory`	Get argument category for Simulink model port from model-specific C function prototype

Function Prototype Control Functions (Continued)

Function	Description
`getArgName`	Get argument name for Simulink model port from model-specific C function prototype
`getArgPosition`	Get argument position for Simulink model port from model-specific C function prototype
`getArgQualifier`	Get argument type qualifier for Simulink model port from model-specific C function prototype
`getDefaultConf`	Get default configuration information for model-specific C function prototype from Simulink model to which it is attached
`getFunctionName`	Get function name from model-specific C function prototype
`getNumArgs`	Get number of function arguments from model-specific C function prototype
`runValidation`	Validate model-specific C function prototype against Simulink model to which it is attached
`setArgCategory`	Set argument category for Simulink model port in model-specific C function prototype
`setArgName`	Set argument name for Simulink model port in model-specific C function prototype
`setArgPosition`	Set argument position for Simulink model port in model-specific C function prototype
`setArgQualifier`	Set argument type qualifier for Simulink model port in model-specific C function prototype
`setFunctionName`	Set function name in model-specific C function prototype

Sample M Script for Configuring a model_step Function Prototype

The following sample M script launches the demo model `rtwdemo_counter` and performs these steps:

1 Calls `set_param` to select `ert.tlc` as the model's system target file.

2. Invokes RTW.ModelSpecificCPrototype to create a new model-specific C function prototype.
3. Calls addArgConf to add argument configuration information for the model's Input and Output ports.
4. Calls attachToModel to attach the function prototype to the loaded model.
5. Further modifies the function prototype using setFunctionName, setArgPosition, setArgCategory, setArgName, and setArgQualifier.
6. Calls runValidation to validate the function prototype against the model to which it is attached.
7. If validation succeeds, calls rtwbuild to invoke the Real-Time Workshop build procedure and generate code.

```
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Create a model-specific C function prototype
a=RTW.ModelSpecificCPrototype

%% Add argument configuration information for Input and Output ports
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

%% Attach the model-specific C function prototype to the model
attachToModel(a,gcs)

%% Rename the step function and change some argument attributes
setFunctionName(a,'StepFunction')
setArgPosition(a,'Output',1)
setArgCategory(a,'Input','Value')
setArgName(a,'Input','InputArg')
setArgQualifier(a,'Input','none')

%% Validate the function prototype against the model
[status,message]=runValidation(a)

%% if validation succeeded, generate code and build
```

```
if status
    rtwbuild(gcs)
end
```

Configuring a Step Function Prototype for a Nonvirtual Subsystem

You can control step function prototypes for nonvirtual subsystems in ERT-based Simulink models, if you generate subsystem code using right-click build. Function prototype control is supported for the following types of nonvirtual subsystems:

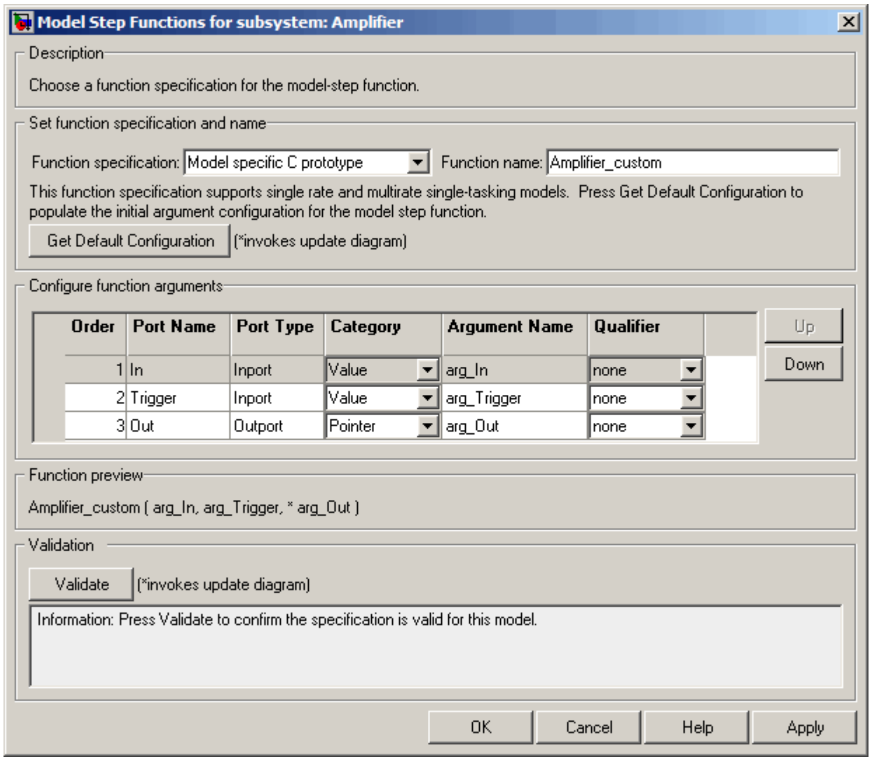
- Triggered subsystems
- Enabled subsytems
- Enabled trigger subsystems
- While subsystems
- For subsystems
- Stateflow subsystems if atomic
- Embedded MATLAB™ subsystems if atomic

To launch the Model Step Functions for subsystem dialog box, open the containing model and invoke the function `RTW.configSubsystemBuild`:

```
RTW.configSubsystemBuild('model/subsystem')
RTW.configSubsystemBuild(gcb)
```

You supply the full block path to the subsystem, such as `'rtwdemo_counter/Amplifier'`, or simply select the subsystem you want to configure and pass the MATLAB function `gcb`, which returns the full block path of the current Simulink block.

The Model Step Functions dialog box for modifying the `Model-specific C prototype` function for the subsystem `rtwdemo_counter/Amplifier` is displayed as follows:



Description

Choose a function specification for the model-step function.

Set function specification and name

Function specification: Model specific C prototype   Function name: Amplifier_custom

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model step function.

Get Default Configuration (*invokes update diagram)

Configure function arguments

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	In	Inport	Value	arg_In	none
2	Trigger	Inport	Value	arg_Trigger	none
3	Out	Outport	Pointer	arg_Out	none

Function preview

Amplifier_custom (arg_In, arg_Trigger, * arg_Out)

Validation

Validate (*invokes update diagram)

Information: Press Validate to confirm the specification is valid for this model.

Right-click building the subsystem will generate the step function according to the customizations you make.

Verifying Generated Code for Customized Step Functions

You can use software-in-the-loop (SIL) testing to verify the generated code for your customized step functions. This involves generating an ERT S-function wrapper for your generated code, which then can be integrated into a Simulink model to verify that the generated code provides the same result as the original model or nonvirtual subsystem. For more information, see "Automatic S-Function Wrapper Generation" on page 2-56 and "Verifying Generated Code with Software-in-the-loop Testing" on page 2-60.

model_step Function Prototype Control Limitations

The following limitations apply to the `model_step` function prototype control capability:

- Function prototype control currently is supported only for step functions generated from a Simulink model.
- The **Generate reusable code** option (on the **Interface** pane of the Configuration Parameters dialog box) must be cleared.
- The **Single output/update function** option (on the **Interface** pane of the Configuration Parameters dialog box) must be selected.
- Function prototype control does not support multitasking models. Multirate models are supported, but must be configured as single-tasking.
- Root-level inports and outports cannot use custom storage classes other than `Auto`.
- The generated code for a parent model does not call the function prototype control step functions generated from referenced models.
- Function prototype control should not be used with the static `ert_main.c` provided by The MathWorks. Specifying a function prototype control configuration other than the default creates a mismatch between the generated code and the default static `ert_main.c`.
- The data structure of a model's external input is removed unless the value of the external input is used in a subsystem implemented by a nonreusable function.
- The data structure for the model's external output is removed except when MAT-file logging is enabled, or if the sample time of the outport is constant.
- An element of the data structure for the model's external output *must be* retained for every outport that does not execute at the fundamental base rate.

Creating and Using Host-Based Shared Libraries

In this section...

Overview

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

Code generation for the `ert_shrlib.tlc` target exports

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

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

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

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

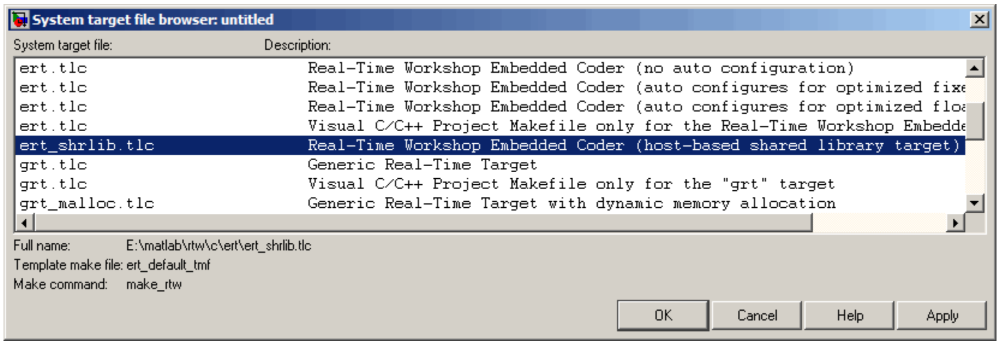
1 Generate a shared library version of your model code

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

Generating a Shared Library Version of Your Model Code

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

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



System target file browser: untitled

System target file:	Description:
ert.tlc	Real-Time Workshop Embedded Coder (no auto configuration)
ert.tlc	Real-Time Workshop Embedded Coder (auto configures for optimized fixe
ert.tlc	Real-Time Workshop Embedded Coder (auto configures for optimized floa
ert.tlc	Visual C/C++ Project Makefile only for the Real-Time Workshop Embedde
ert_shrlib.tlc	Real-Time Workshop Embedded Coder (host-based shared library target)
grt.tlc	Generic Real-Time Target
grt.tlc	Visual C/C++ Project Makefile only for the "grt" target
grt_malloc.tlc	Generic Real-Time Target with dynamic memory allocation

```
Full name:          E:\matlab\rtw\c\ert\ert_shrlib.tlc
Template make file: ert_default_tmf
Make command:       make_rtw
```

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

2 Build the model.

3 After the build completes, you can examine the generated code in the model subdirectory, and the `.dll` file or `.so` file that has been generated into your current directory.

Creating Application Code to Load and Use Your Shared Library File

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

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

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

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

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

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

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

Example Application Header File

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

```
#ifndef _APP_MAIN_HEADER_
#define _APP_MAIN_HEADER_

typedef struct {
    int32_T Input;
} ExternalInputs_rtwdemo_shrlib;

typedef struct {
    int32_T Output;
} ExternalOutputs_rtwdemo_shrlib;

#endif /*_APP_MAIN_HEADER_*/
```

Example Application C Code

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

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

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

#endif

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

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

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

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

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

```
    mdl_initialize =(void(*)(boolean_T))GETSYMBOLADDR(handleLib ,
                      "rtwdemo_shrlib_initialize");
    mdl_step       =(void(*)(void))GETSYMBOLADDR(handleLib ,
                      "rtwdemo_shrlib_step");
    mdl_terminate  =(void(*)(void))GETSYMBOLADDR(handleLib ,
                      "rtwdemo_shrlib_terminate");
    mdl_Uptr       =(ExternalInputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
                      "rtwdemo_shrlib_U");
    mdl_Yptr       =(ExternalOutputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
                      "rtwdemo_shrlib_Y");
    sum_outptr     =(uint8_T*)GETSYMBOLADDR(handleLib , "sum_out");
#endif

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

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

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

Example Application M Script

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

Host-Based Shared Library Limitations

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

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

  You can change the default behavior and retain the `.lib` file by modifying the corresponding template makefile (TMF). If you do this, be aware that the generated *`model`*`.h` file will need a small modification to be used together with the generated `ert_main.c` for implicit linking. For example, if you are using Visual C++, you will need to declare `__declspec(dllimport)` in front of all data to be imported implicitly from the shared library file.
- To reconstruct a model simulation using a generated host-based shared library, the application author must maintain the timing between system and shared library function calls in the original application. The timing needs to be consistent to ensure correct simulation and integration results.

Custom Storage Classes

Introduction to Custom Storage Classes

In this section...

Overview

In Real-Time Workshop, the *storage class* specification of a signal, tunable parameter, block state, or data object specifies how that entity is declared, stored, and represented in generated code.

Note that in the context of Real-Time Workshop, the term "storage class" is not synonymous with the term "storage class specifier", as used in the C language.

Real-Time Workshop defines built-in storage classes for use with all targets. Examples of built-in storage classes are `Auto`, `ExportedGlobal`, and `ImportedExtern`. These storage classes provide limited control over the form of the code generated for references to the data. For example, data of storage class `Auto` is typically declared and accessed as an element of a structure, while data of storage class `ExportedGlobal` is declared and accessed as unstructured global variables. Built-in storage classes are discussed in detail in the "Working with Data Structures" section of the Real-Time Workshop documentation.

The built-in storage classes are suitable for many applications, but embedded system designers often require greater control over the representation of data. For example, you may need to

- Define structures for storage of parameter or signal data.
- Conserve memory by storing Boolean data in bit fields.
- Integrate generated code with legacy software whose interfaces cannot be modified.
- Generate data structures and definitions that comply with your organization's software engineering guidelines for safety-critical code.

Real-Time Workshop Embedded Coder's *custom storage classes* (CSCs) provide extended control over the constructs required to represent data in an embedded algorithm. CSCs extend the built-in storage classes provided by Real-Time Workshop. Real-Time Workshop Embedded Coder provides

- A set of ready-to-use CSCs. These CSCs are designed to be useful in code generation for embedded systems development. CSC functionality is integrated into the `Simulink.Signal` and `Simulink.Parameter` classes; you do not need to use special object classes to generate code with CSCs.

  If you are unfamiliar with the `Simulink.Signal` and `Simulink.Parameter` classes and objects, you should read the "Simulink Data Objects and Code Generation" section of the Real-Time Workshop documentation.
- The Custom Storage Class Designer (`cscdesigner`) tool. This tool lets you define additional CSCs that are tailored to your code generation requirements. The Custom Storage Class Designer provides a graphical user interface that lets you implement CSCs quickly and easily. You can use your CSCs in code generation immediately, without any Target Language Compiler (TLC) or other programming.
- The Simulink Data Class Designer. This chapter describes how to use the Simulink Data Class Designer to create a data object package and associate your own custom CSC definitions with classes contained in the package. For a general description of the Simulink Data Class Designer, see the Simulink documentation.

The demo `rtwdemo_importstruct` shows how the Custom Storage Class capabilities work together. Using the techniques shown in the demo, you can specify and change the values of any number of parameters with a single C statement. The statement assigns to a pointer variable the address of a structure that defines the desired values. Assigning the address of a different structure changes all the values at once.

Custom Storage Class Memory Sections

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

See “Editing Memory Section Definitions” on page 3-31 for details about using the Custom Storage Class designer to define memory sections. More information about memory sections appears in Chapter 4, “Memory Sections”.

Custom Storage Classes and Simulink Data Objects

In this section...

Overview

CSCs are associated with Simulink data class packages (such as the `Simulink` package) and with classes within packages (such as the `Simulink.Parameter` and `Simulink.Signal` classes). The custom storage classes associated with a package are defined by a *CSC registration file*.

A CSC registration file is provided for the `Simulink` package. This registration file provides predefined CSCs for use with the `Simulink.Signal` and `Simulink.Parameter` classes (and with subclasses derived from these classes). The predefined CSCs are sufficient for a wide variety of applications.

If you use only predefined CSCs, you do not need to be concerned with CSC registration files. If you want to customize or extend the predefined CSCs, or create CSCs for use with data class packages other than the `Simulink` package, you can by using the Custom Storage Class Designer. The Custom Storage Class Designer is described in "Designing Custom Storage Classes and Memory Sections" on page 3-16.

The next three sections discuss topics related to predefined CSCs and their use in code generation:

- "Predefined CSCs" on page 3-7 discusses the ready-to-use CSCs provided for parameter and signal objects.
- "Setting Custom Storage Class Properties" on page 3-10 demonstrates how to configure the CSC-related properties of parameter and signal objects.
- "Generating Code with CSCs" on page 3-11 guides you through the steps required to generate code using CSCs, using signal objects as an example.

Predefined CSCs

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

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

- `CustomStorageClass`: To assign a custom storage class to a signal or parameter object, you set the `RTWInfo.CustomStorageClass` property to one of the available CSC names and `RTWInfo.StorageClass` to `Custom`. Summary of Predefined Simulink CSCs for Signal and Parameter Objects on page 3-8 lists the predefined set of CSCs provided by Real-Time Workshop Embedded Coder.
- `CustomAttributes`: Some CSCs have *instance-specific* properties that define attributes of individual objects (or instances) of that class. The `RTWInfo.CustomAttributes` property lets you define these attributes. For example, you can pack signal objects of class `Struct` into different data structures in the generated code by setting the `RTWInfo.CustomAttributes.StructName` property for each object. Summary of Instance-Specific Properties for CSCs on page 3-9 lists instance-specific properties for the predefined set of CSCs provided by Real-Time Workshop Embedded Coder.

Note that some CSCs are valid for parameter objects but not signal objects and vice versa (even though they are not defined in predefined CSCs). For example, you can assign the storage class `Const` to a parameter object. This storage class is not valid for signals, because, in general, signal data is not constant. Summary of Predefined Simulink CSCs for Signal and Parameter Objects on page 3-8 indicates whether each class is valid for parameter or signal objects.

Summary of Predefined Simulink CSCs for Signal and Parameter Objects

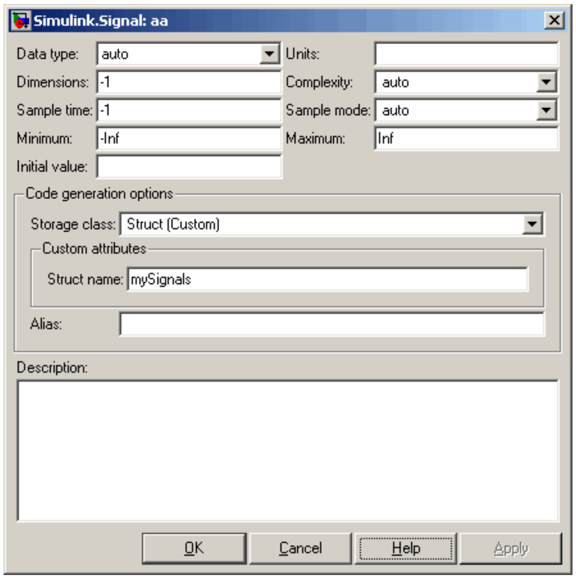
Class Name (Enumerated)	Available for Signals	Available for Parameters	Purpose
`BitField`	Y	Y	Generate a `struct` declaration that embeds Boolean data in named bit fields.
`Const`	N	Y	Generate a constant declaration with the `const` type qualifier.
`ConstVolatile`	N	Y	Generate declaration of volatile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

Summary of Instance-Specific Properties for CSCs

Class Name (Enumerated)	Instance-Specific Property	Purpose
`BitField`	`CustomAttributes.StructName`	Name of the bitfield `struct` into which the code generator packs the object's Boolean data.
`ExportToFile`	`CustomAttributes.HeaderFile`	Name of header (`.h`) file that contains exported variable declarations and export directives for the object.
`ImportFromFile`	`CustomAttributes.HeaderFile`	Name of header (`.h`) file containing global variable declarations the code generator imports for the object.
`Struct`	`CustomAttributes.StructName`	Name of the `struct` into which the code generator packs the object's data.

Setting Custom Storage Class Properties

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


Simulink.Signal: aa

Data type: auto
Units:
Dimensions: -1
Complexity: auto
Sample time: -1
Sample mode: auto
Minimum: -Inf
Maximum: Inf
Initial value:

Code generation options
Storage class: Struct (Custom)
Custom attributes
Struct name: mySignals
Alias:

Description:

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

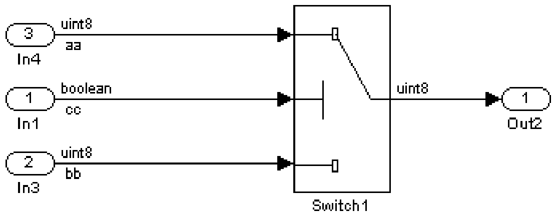
```
aa = Simulink.Signal;
aa.RTWInfo.StorageClass = 'Custom';
aa.RTWInfo.CustomStorageClass = 'Struct';
aa.RTWInfo.CustomAttributes.StructName = 'mySignals';
```

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

In the generated code, storage for the signal `aa` is allocated within a `struct` named `mySignals`. This is demonstrated in the next section, "Generating Code with CSCs" on page 3-11.

Generating Code with CSCs

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



This example uses signal objects, but the procedure for generating code from parameter objects (or from any class of objects that supports CSCs) is almost the same. (If you plan to use CSCs with parameter objects, see "Setting Code Generation Options for Custom Storage Classes" on page 3-50 for the correct use of the **Inline parameters** option.)

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

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

Set Model Properties

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

Instantiate Signal Objects

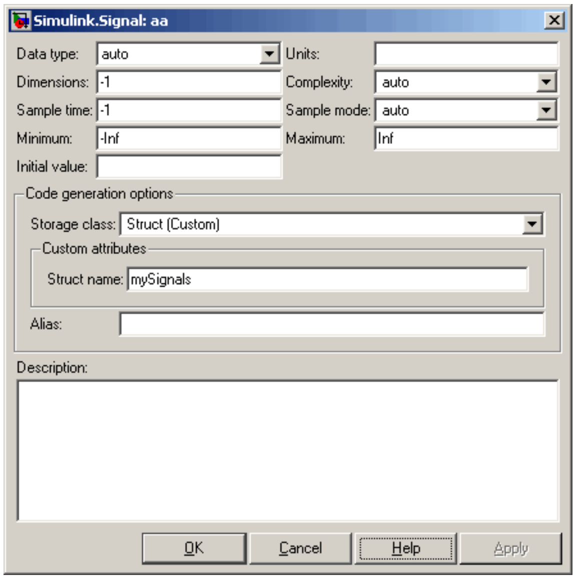
The next step is to instantiate signal objects. You can do this with MATLAB commands as shown below.

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

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

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

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



Data type:	auto	Units:	
Dimensions:	-1	Complexity:	auto
Sample time:	-1	Sample mode:	auto
Minimum:	-Inf	Maximum:	Inf
Initial value:			

Code generation options

Storage class: Struct (Custom)

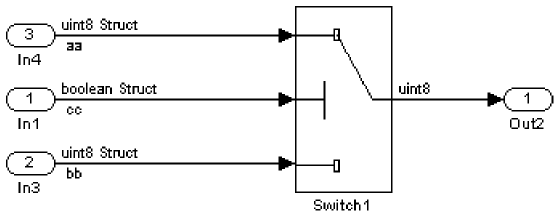
Custom attributes

Struct name: mySignals

Alias:

Description:

The association between identically named model signals and signal objects is formed automatically. The symbols `aa`, `bb`, and `cc` resolve to the signal objects `aa`, `bb`, and `cc`, which have custom storage class `Struct`. You can display the storage class of the signals in the block diagram by selecting **Port/Signal Display > Storage Class** from the Simulink **Format** menu. The figure below shows the block diagram with signal data types and signal storage classes displayed.



Generate Code. The model is now configured to generate the desired data structure for the signals. After code generation, the relevant definitions and declarations are located in three files:

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

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

- `model.c` or `.cpp` declares the variable `mySignals`, as specified in the object's instance-specific `StructName` attribute. The variable is referenced in the code generated for the Switch block.

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

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

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

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

  extern mySignals_type mySignals;
  ```

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

Designing Custom Storage Classes and Memory Sections

In this section...

Using the Custom Storage Class Designer

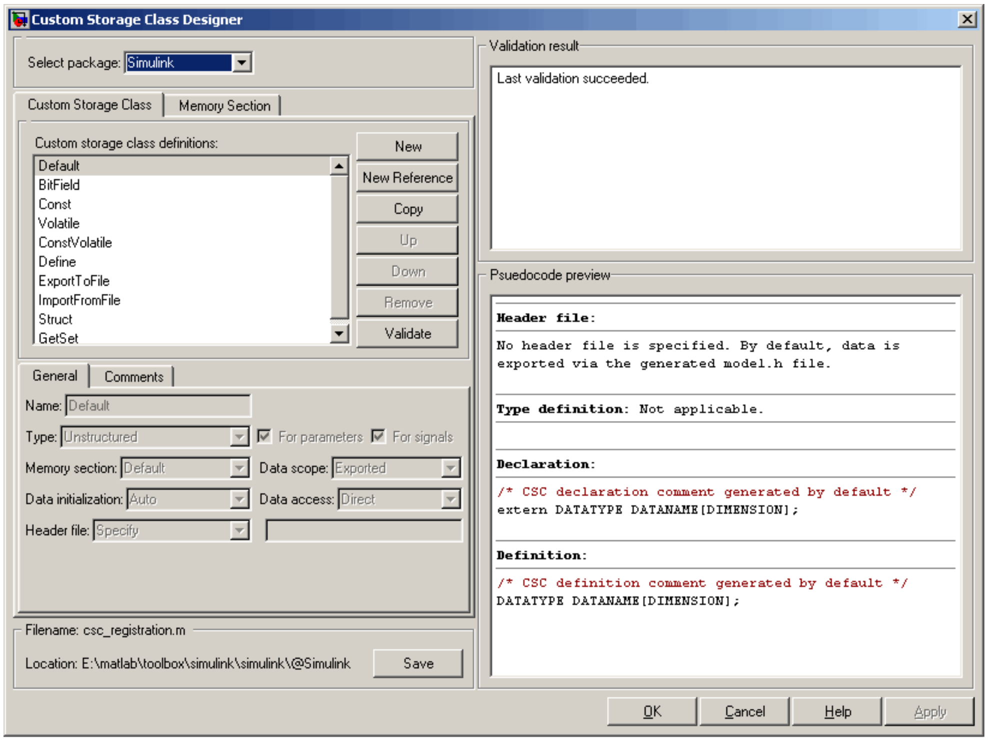
The Custom Storage Class Designer (`cscdesigner`) is a tool for creating and managing custom storage classes and memory sections. You can use the Custom Storage Class Designer to:

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

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

```
cscdesigner
```

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



Select package: Simulink

Custom storage class definitions:

Default
BitField
Const
Volatile
ConstVolatile
Define
ExportToFile
ImportFromFile
Struct
GetSet

Name: Default

Type: Unstructured ✓ For parameters ✓ For signals

Memory section: Default Data scope: Exported

Data initialization: Auto Data access: Direct

Header file: Specify

Filename: csc_registration.m

Location: E:\matlab\toolbox\simulink\simulink\@Simulink

Validation result

Last validation succeeded.

Psuedocode preview

```
Header file:

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

Type definition: Not applicable.

Declaration:

/* CSC declaration comment generated by default */
extern DATATYPE DATANAME[DIMENSION];

Definition:

/* CSC definition comment generated by default */
DATATYPE DATANAME[DIMENSION];
```

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

- **Select package**: Lets you select from a menu of data class packages that have CSC definitions associated with them. See “Selecting a Data Class Package” on page 3-18 for details.
- **Custom Storage Class / Memory Section** properties: Lets you select, view, edit, copy, verify, and perform other operations on CSC definitions or memory section definitions. The common controls in the **Custom Storage Class / Memory Section** properties panel are described in “Selecting and Editing CSCs, Memory Sections, and References” on page 3-19.

- When the **Custom Storage Class** tab is selected, you can select a CSC definition or reference from a list and edit its properties. See "Editing Custom Storage Class Properties" on page 3-21 for details.
- When the **Memory Section** tab is selected, you can select a memory section definition or reference from a list and edit its properties. See "Editing Memory Section Definitions" on page 3-31 for details.

- **Filename**: Displays the filename and location of the current CSC registration file, and lets you save your CSC definition to that file. See "Saving Your Definitions" on page 3-20 for details.
- **Pseudocode preview**: Displays a preview of code that is generated from objects of the given class. The preview is pseudocode, since the actual symbolic representation of data objects is not available until code generation time. See "Previewing Generated Code" on page 3-33 for details.
- **Validation result**: Displays any errors encountered when the currently selected CSC definition is validated. See "Validating CSC Definitions" on page 3-28 for details.

Selecting a Data Class Package

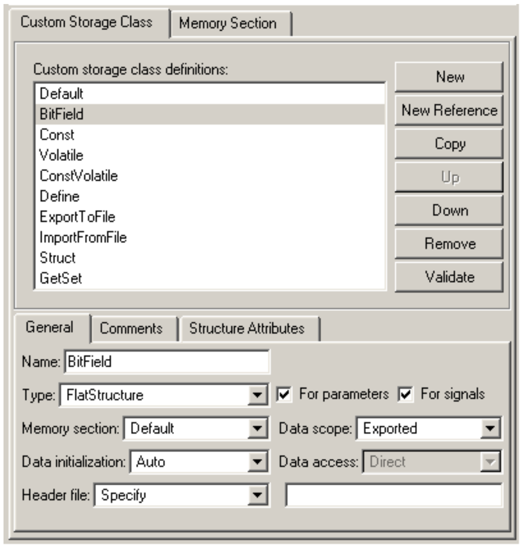
A CSC or memory section definition or reference is uniquely associated with a Simulink data class package. The link between the definition/reference and the package is formed when a CSC registration file (`csc_registration.m`) is located in the package directory. You never need to search for or edit a CSC registration file directly: the Custom Storage Class Designer locates all available CSC registration files and displays the associated package names in the **Select package** panel:

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

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

Selecting and Editing CSCs, Memory Sections, and References

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



Name: BitField
Type: FlatStructure ✓ For parameters ✓ For signals
Memory section: Default Data scope: Exported
Data initialization: Auto Data access: Direct
Header file: Specify

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

- "Editing Custom Storage Class Properties" on page 3-21 for information about the properties of the predefined CSCs.
- "Editing Memory Section Definitions" on page 3-31 for information about the properties of the predefined memory section definitions.

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

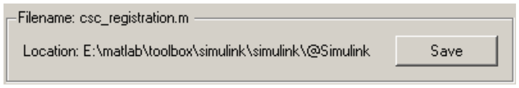
- **New**: Creates a new CSC or memory section with default values.
- **New Reference:** Creates a reference to a CSC or memory section definition in another package. The default initially has a default name and properties. See:
- **Copy**: Creates a copy of the selected definition / reference. Copies are given a default name using the convention:

  ```
  definition_name_n
  ```

  where `definition_name` is the name of the original definition, and `n` is an integer indicating successive copy numbers (for example: `BitField_1`, `BitField_2`, ...)
- **Up**: Moves the selected definition one position up in the list.
- **Down**: Moves the selected definition one position down in the list
- **Remove**: Removes the selected definition from the list.
- **Validate**: Performs a consistency check on the currently selected definition. Errors are reported in the **Validation result** panel.

Saving Your Definitions

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


Filename: csc_registration.m

Location: E:\matlab\toolbox\simulink\simulink\@Simulink

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

Note If you edit a CSC definition (including any associated memory section) that has been assigned to existing parameter or signal objects, you must restart MATLAB after editing and saving the CSC definition.

Editing Custom Storage Class Properties

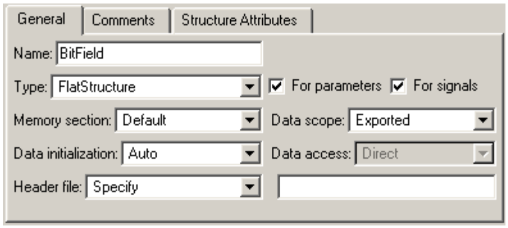
To view and edit the properties of a CSC, click the **Custom Storage Class** tab in the **Custom Storage Class / Memory Section** panel. Then, select a CSC name from the **Custom storage class definitions** list.

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

The property categories and corresponding tabs are as follows:

General

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



Name: BitField
Type: FlatStructure ✓ For parameters ✓ For signals
Memory section: Default Data scope: Exported
Data initialization: Auto Data access: Direct
Header file: Specify

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

- **Name**: The CSC name, selected from the **Custom storage class definitions** list.

- **Type**: Specifies how objects of this class are stored. Values:
  - `Unstructured`: Objects of this class generate unstructured storage declarations (for example, scalar or array variables), for example:

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

  - `FlatStructure`: Objects of this class are stored as members of a struct. A **Structure Attributes** tab is also displayed, allowing you to specify additional properties such as the struct name. See "Structure Attributes" on page 3-26.
  - `Other`: Used for certain data layouts, such as nested structures, that cannot be generated using the standard `Unstructured` and `FlatStructure` custom storage class types. If you want to generate other types of data, you can create a new custom storage class from scratch by writing the necessary TLC code. See "Defining Advanced Custom Storage Class Types" on page 3-42 for more information.
- **For parameters** and **For signals**: These options let you enable a CSC for use with only certain classes of data objects. For example, it does not make sense to assign storage class `Const` to a `Simulink.Signal` object. Accordingly, the **For signals** option for the `Const` class is deselected, while the **For parameters** is selected.
- **Memory section**: Selects one of the memory sections defined in the **Memory Section** panel. See "Editing Memory Section Definitions" on page 3-31.
- **Data scope**: Controls the scope of symbols generated for data objects of this class. Values:
  - `Auto`: Symbol scope is determined internally by Real-Time Workshop. If possible, symbols have `File` scope. Otherwise, they have `Exported` scope.
  - `Exported`: Symbols are exported to external code in the header file specified by the **Header File** field. If no **Header File** is specified, symbols are exported to external code in *`model`*`.h`.
  - `Imported`: Symbols are imported from external code in the header file specified by the **Header File** field. If you do not specify a header file, an `extern` directive is generated in *`model`*`_private.h`. For imported data, if the **Data initialization** value is `Macro` , a header file *must* be specified.

- `File`: The scope of each symbol is the file that defines it. File scope requires each symbol to be used in a single file. If the same symbol is referenced in multiple files, an error occurs at code generation time.
- `Instance specific`: Symbol scope is defined by the **Data scope** property of each data object.

• **Data initialization**: Controls how storage is initialized in generated code. Values:

- `Auto`: Storage initialization is determined internally by Real-Time Workshop. Parameters have `Static` initialization, and signals have `Dynamic` initialization.
- `None`: No initialization code is generated.
- `Static`: A static initializer of the following form is generated:

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

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

```
#define data numeric_value
```

  The `Macro` initialization option is available only for use with unstructured parameters. It is not available when the class is configured for generation of structured data, or for signals. If the **Data scope** value is `Imported`, a header file *must* be specified.
- `Instance specific`: Initialization is defined by the **Data initialization** property of each data object.

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

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

  - `Direct`: Symbols are declared as simple variables, such as

    ```
    extern myType myVariable;
    ```

  - `Pointer`: Symbols are declared as pointer variables, such as

    ```
    extern myType *myVariable;
    ```

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

- **Header file**: Defines the name of a header file that contains exported or imported variable declarations and definitions for objects of this class. Values:

  - `Specify`: An edit field is displayed to the right of the property. This lets you specify a header file for exported or imported storage declarations. Specify the full filename, including the filename extension (such as `.h`). Use quotes or brackets as in C code to specify the location of the header file. Leave the edit field empty to specify no header file.

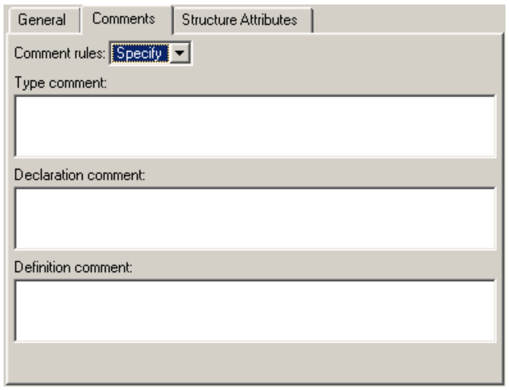
  - `Instance specific`: The header file for each data object is defined by the **Header file** property of the object. Leave the property undefined to specify no header file for that object.

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

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

Comments

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



Comment rules: Specify

Type comment:

Declaration comment:

Definition comment:

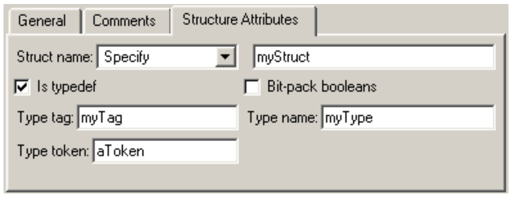
Comments must conform to the ANSI C standard (`/*...*/`). Use `\n` to specify a new line.

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

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

Structure Attributes

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



Struct name: Specify — myStruct
[✓] Is typedef
[ ] Bit-pack booleans
Type tag: myTag
Type name: myType
Type token: aToken

Structure Attributes Panel

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

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

  If you select `Specify`, an edit field appears (as shown in Structure Attributes Panel on page 3-26) for entry of the name of the structure to be used in the `struct` definition. Edit fields **Type tag**, **Type token**, and **Type name** are also displayed.

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

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

  Otherwise, a simple struct definition is generated.

- **Bit-pack booleans**: When this option is selected, signals and/or parameters that have Boolean data type are packed into bit fields in the generated struct.

- **Type tag**: Specifies a tag to be generated after the `struct` keyword in the struct definition.

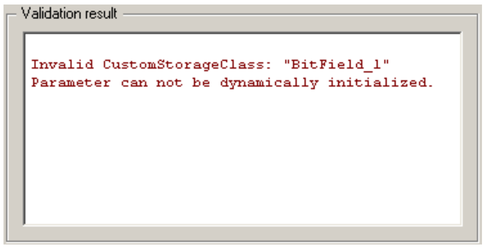
- **Type token**: Some compilers support an additional token (which is simply another string) after the type tag. To generate such a token, enter the string in this field.
- **Type name**: Specifies the string to be used in `typedef` definitions. This field is visible if **Is typedef** is selected.

The following listing is the pseudocode preview corresponding to the **Structure Attributes** properties displayed in Structure Attributes Panel on page 3-26.

```
Header file:

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


Type definition:

/* CSC type comment generated by default */

typedef struct aToken myTag {
   :
} myType;


Declaration:

/* CSC declaration comment generated by default */

extern myType MyStruct;


Definition:

/* CSC definition comment generated by default */

myType MyStruct = {...};
```

Validating CSC Definitions

To validate a CSC definition, select the definition on the **Custom Storage Class** panel and click **Validate** . The Custom Storage Class Designer then checks the definition for consistency. The **Validation result** panel displays any errors encountered when the selected CSC definition is validated. The next figure shows the **Validation result** panel with a typical error message:



```
Invalid CustomStorageClass: "BitField_1"
Parameter can not be dynamically initialized.
```

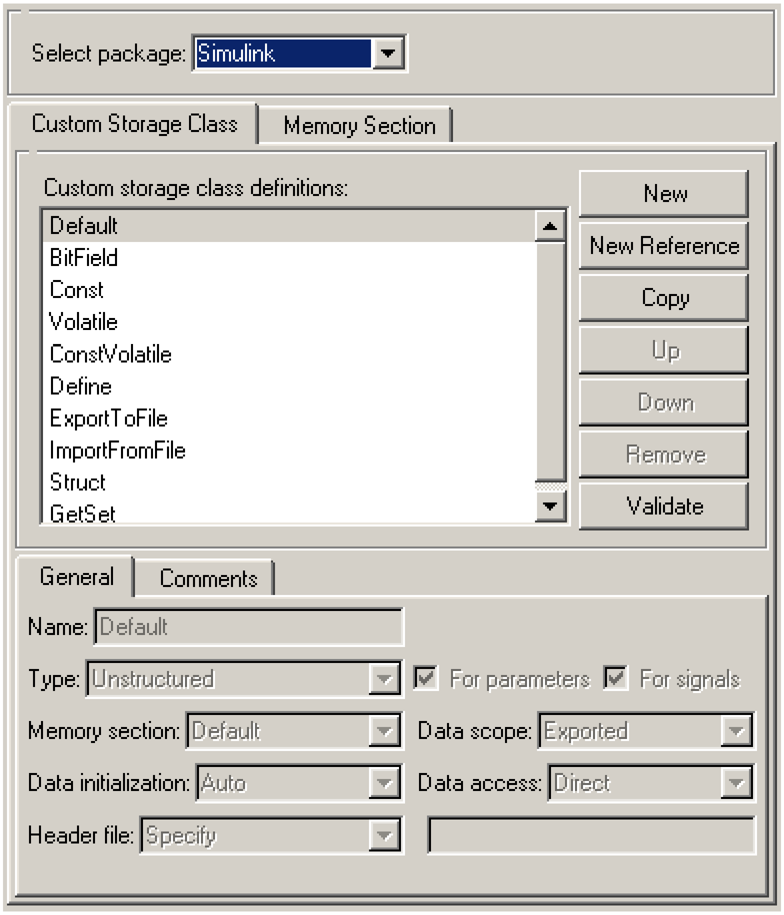
Validation is also performed whenever CSC definitions are saved. In this case, all CSC definitions are selected. (See “Saving Your Definitions” on page 3-20.)

Using Custom Storage Class References

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

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

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



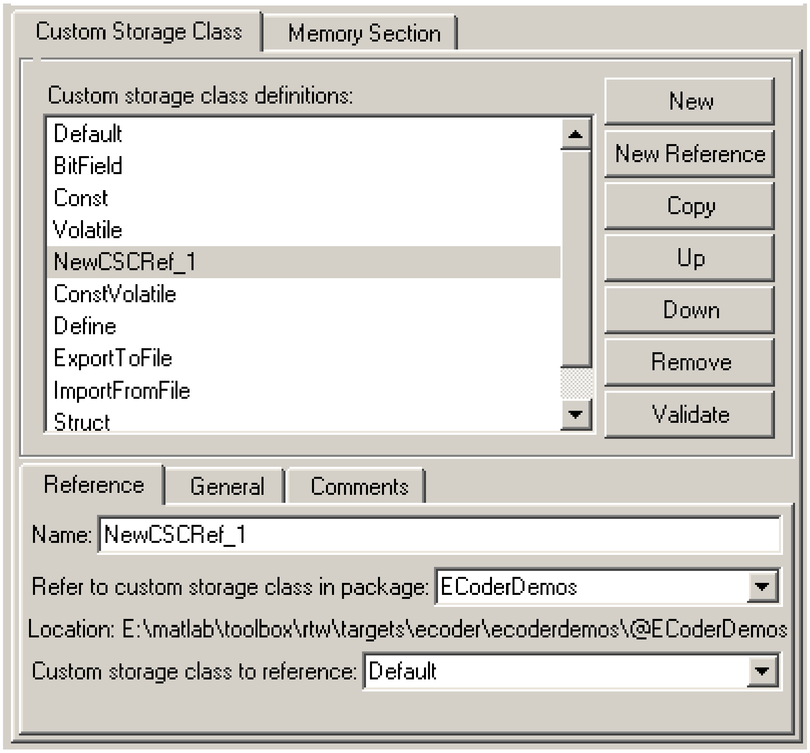
2 Select the **Custom Storage Class** tab.

3 Use **Select Package** to select the package in which you want to reference a class or section defined in some other package.

4 In the **Custom storage class definitions** pane, select the existing definition below which you want to insert the reference.

5 Click **New Reference**.

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



Name: NewCSCRef_1

Refer to custom storage class in package: ECoderDemos

Location: E:\matlab\toolbox\rtw\targets\ecoder\ecoderdemos\@ECoderDemos

Custom storage class to reference: Default

6 Use the **Name** field to enter a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package.

7 Set **Refer to custom storage class in package** to specify the package that contains the custom storage class you want to reference.

8 Set **Custom storage class to reference** to specify the custom storage class to be referenced. Trying to create a circular reference generates an error and leaves the package unchanged.

9 Click **OK** or **Apply** to save the changes to memory. See "Saving Your Definitions" on page 3-20 for information about saving changes permanently.

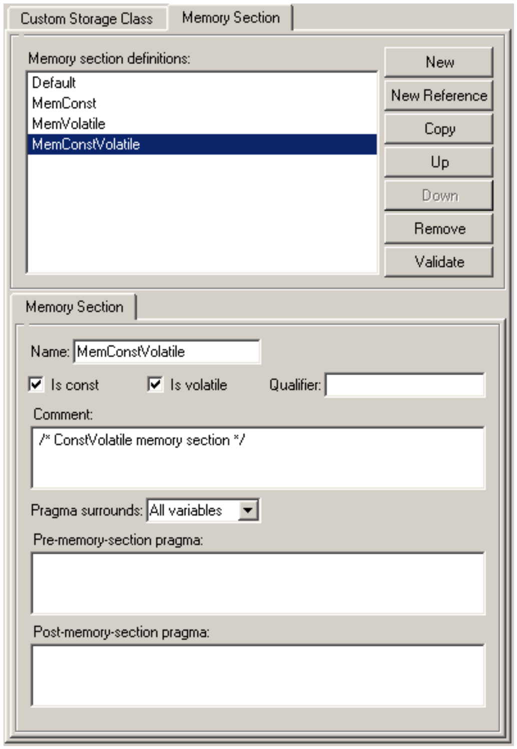
You can use Custom Storage Class Designer capabilities to copy, reorder, validate, and otherwise manage classes that have been added to a class by reference. However, you cannot change the underlying definitions. You can change a custom storage class only in the package where it was originally defined.

Changing Existing CSC References

To change an existing CSC reference, select it in the **Custom storage class definitions** pane. The **Reference** tab appears, showing the current properties of the reference. Make any needed changes, then click **OK** or **Apply** to save the changes to memory. See "Saving Your Definitions" on page 3-20 for information about saving changes permanently.

Editing Memory Section Definitions

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



Memory section definitions:
Default
MemConst
MemVolatile
MemConstVolatile

Memory Section

Name: MemConstVolatile

[✓] Is const [✓] Is volatile Qualifier:

Comment:
/* ConstVolatile memory section */

Pragma surrounds: All variables

Pre-memory-section pragma:

Post-memory-section pragma:

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

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

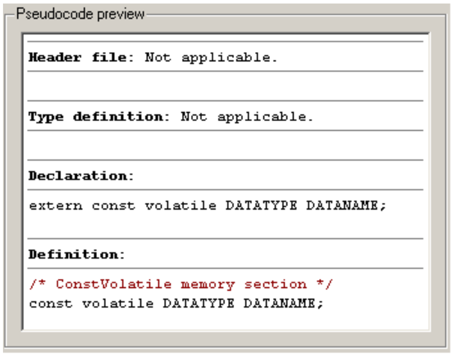
The available memory section definitions also appear in the **Memory section name** menu in the **Custom Storage Class** panel.

The properties of a memory section definition are as follows:

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

Previewing Generated Code

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



Header file: Not applicable.

Type definition: Not applicable.

Declaration:

```
extern const volatile DATATYPE DATANAME;
```

Definition:

```
/* ConstVolatile memory section */
const volatile DATATYPE DATANAME;
```

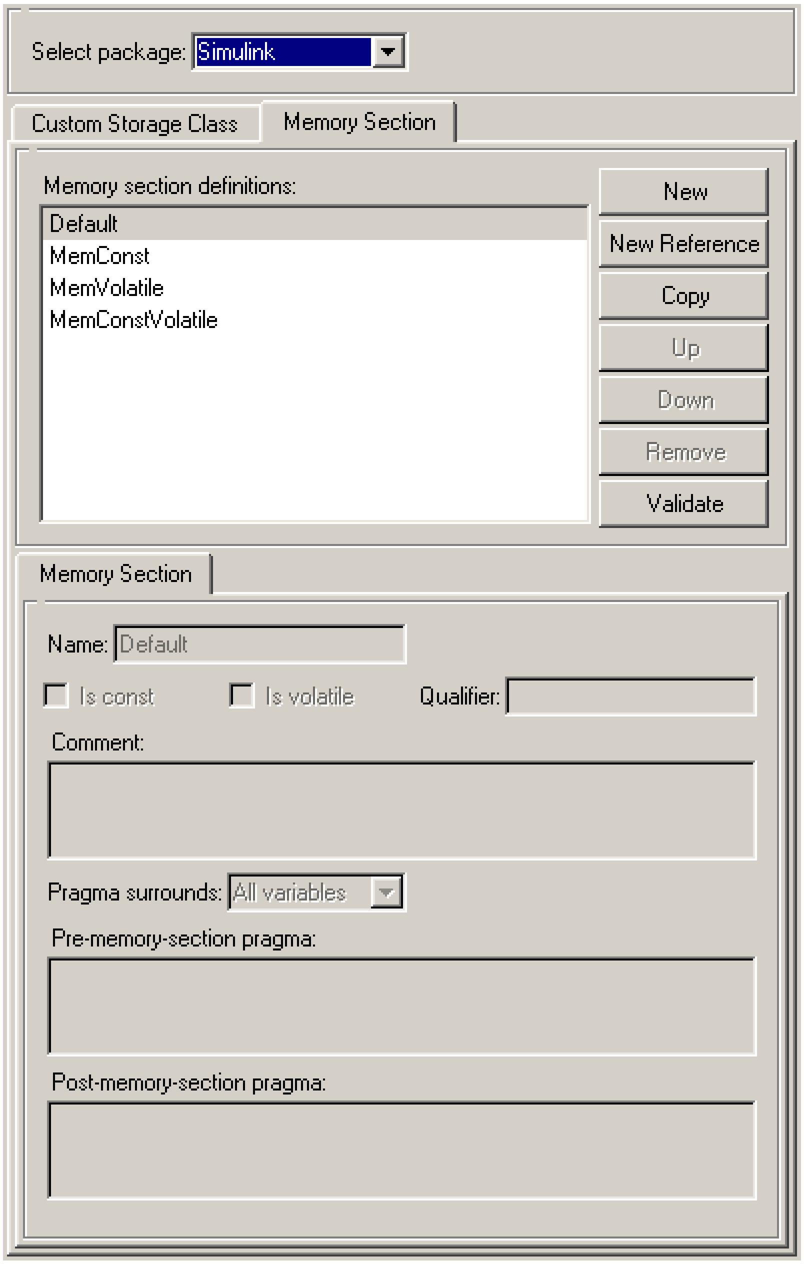
Using Memory Section References

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

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

1 Type `cscdesigner` to launch the Custom Storage Class Designer.

2 Select the **Memory Section** tab. The relevant part of the designer window looks like this:


Select package: Simulink

Memory section definitions:

Default
MemConst
MemVolatile
MemConstVolatile

Name: Default

☐ Is const ☐ Is volatile Qualifier:

Comment:

Pragma surrounds: All variables

Pre-memory-section pragma:

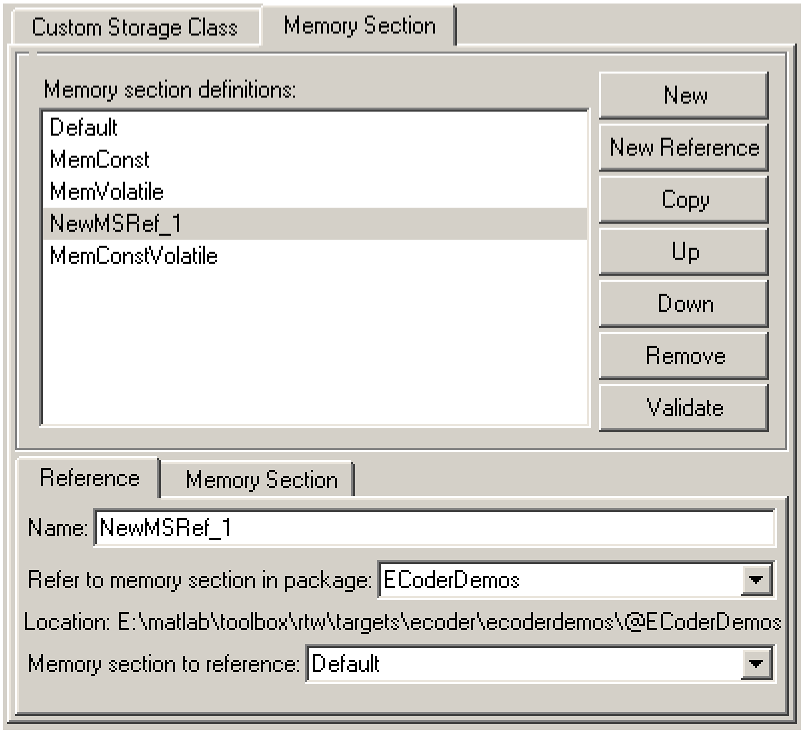
Post-memory-section pragma:

3 Use **Select Package** to select the package in which you want to reference a class or section defined in some other package.

4 In the **Memory section definitions** pane, select the existing definition below which you want to insert the reference.

5 Click **New Reference**.

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



Memory section definitions:
Default
MemConst
MemVolatile
NewMSRef_1
MemConstVolatile

Name: NewMSRef_1

Refer to memory section in package: ECoderDemos

Location: E:\matlab\toolbox\rtw\targets\ecoder\ecoderdemos\@ECoderDemos

Memory section to reference: Default

6 Use the **Name** field to enter a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package.

7. Set **Refer to memory section in package** to specify the package that contains the memory section you want to reference.

8. Set **Memory section to reference** to specify the memory section to be referenced. Trying to create a circular reference generates an error and leaves the package unchanged.

9. Click **OK** or **Apply** to save the changes to memory. See "Saving Your Definitions" on page 3-20 for information about saving changes permanently.

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

Changing Existing Memory Section References

To change an existing memory section reference, select it in the **Memory section definitions** pane. The **Reference** tab appears, showing the current properties of the reference. Make any needed changes, then click **OK** or **Apply** to save the changes to memory. See "Saving Your Definitions" on page 3-20 for information about saving changes permanently.

Creating Packages with CSC Definitions

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

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

   ```
   sldataclassdesigner
   ```

2. The Data Class Designer loads all packages that exist on the MATLAB path.

3. To create a new package, click **New** next to the **Package name** field. If desired, edit the **Package name**. Then, click **OK**.

4. In the **Parent directory** field, enter the path to the directory where you want to store the new package.

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

   ---

5. Click on the **Classes** tab.

6. Create a new class by clicking **New** next to the **Class name** field. If desired, edit the **Class name**. Then, click **OK**.

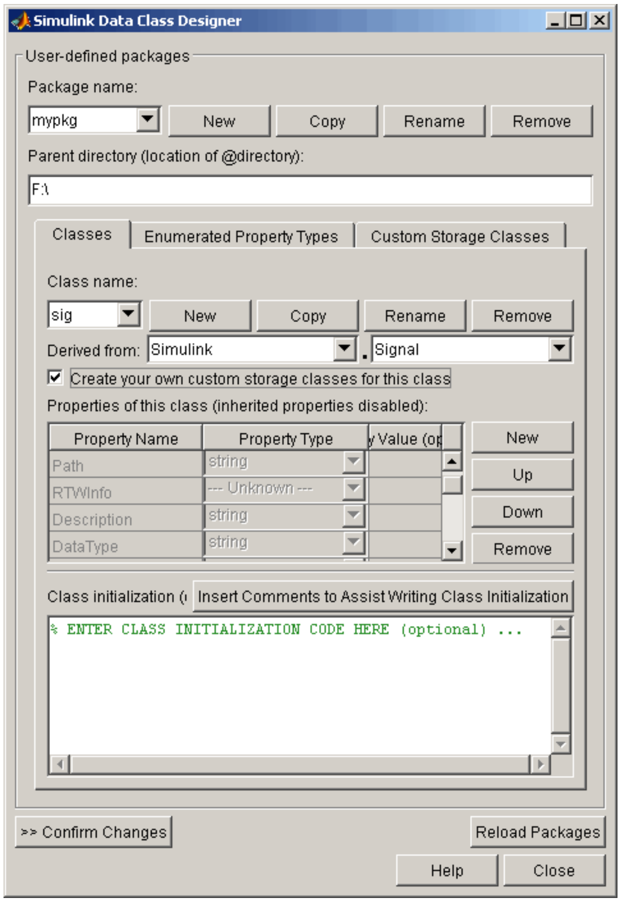
7. In the **Derived from** menus, select `Simulink.Signal` or `Simulink.Parameter`.

8. The **Create your own custom storage classes for this class** option is now enabled. This option is enabled when the selected class is derived from `Simulink.Signal` or `Simulink.Parameter`. You must select this option to create CSCs for the new class. If the **Create your own custom storage**

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

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

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


Simulink Data Class Designer
User-defined packages
Package name:
mypkg
New
Copy
Rename
Remove
Parent directory (location of @directory):
F:\
Classes
Enumerated Property Types
Custom Storage Classes
Class name:
sig
New
Copy
Rename
Remove
Derived from: Simulink . Signal
☑ Create your own custom storage classes for this class
Properties of this class (inherited properties disabled):
New
Up
Down
Remove
Class initialization (
Insert Comments to Assist Writing Class Initialization
>> Confirm Changes
Reload Packages
Help
Close

Property Name	Property Type	y Value (op
Path	string	
RTWInfo	--- Unknown ---	
Description	string	
DataType	string	

```
% ENTER CLASS INITIALIZATION CODE HERE (optional) ...
```

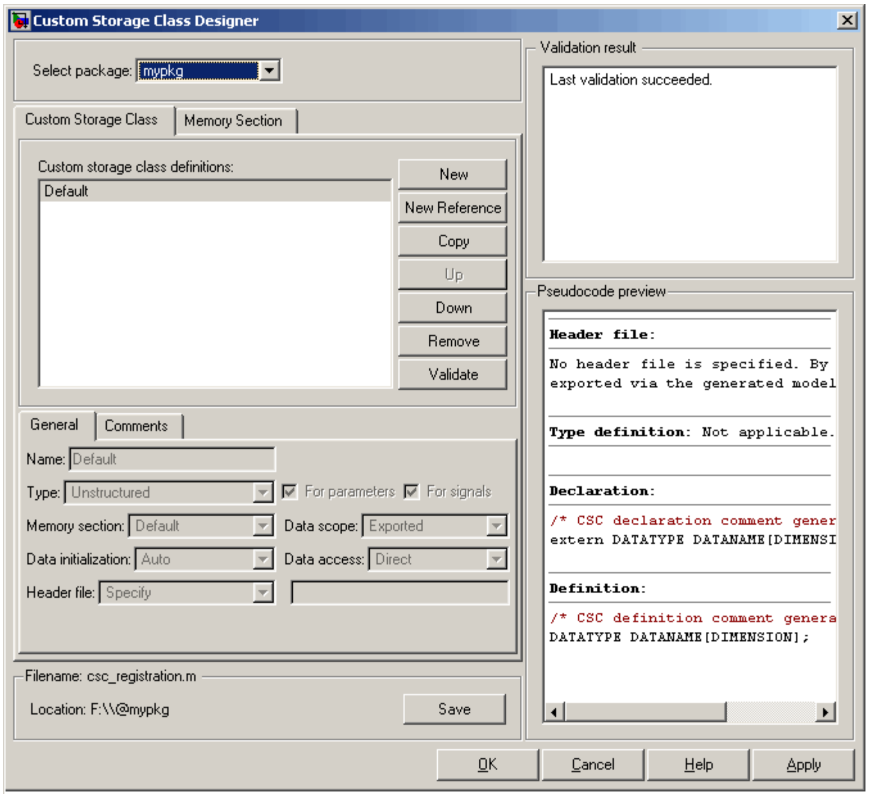
9 If desired, repeat steps 6–8 to add other derived classes to the package and associate CSCs with them.

10 Click **Confirm Changes**. In the **Confirm Changes** pane, select the package you created. Add the parent directory to the MATLAB path if necessary. Then, click **Write Selected**.

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

11 Click **Close**.

12 You can now view and edit the CSCs belonging to your package in the Custom Storage Class Designer. Initially, the package contains only the `Default` CSC definition, as shown in the figure below.



Select package: mypkg

Validation result

Last validation succeeded.

Custom storage class definitions:

Default

Name: Default

Type: Unstructured ☑ For parameters ☑ For signals

Memory section: Default Data scope: Exported

Data initialization: Auto Data access: Direct

Header file: Specify

Filename: csc_registration.m

Location: F:\\@mypkg

Pseudocode preview

```
Header file:

No header file is specified. By
exported via the generated model

Type definition: Not applicable.

Declaration:

/* CSC declaration comment gener
extern DATATYPE DATANAME[DIMENSI

Definition:

/* CSC definition comment genera
DATATYPE DATANAME[DIMENSION];
```

13 Add and edit your CSC and memory section definitions, as described in “Designing Custom Storage Classes and Memory Sections” on page 3-16. After you have created CSC definitions for your package, you can instantiate objects of the classes belonging to your package, and assign CSCs to them.

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

Defining Advanced Custom Storage Class Types

In this section...

Overview

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

The `GetSet` CSC (see "GetSet Custom Storage Class for Data Store Memory" on page 3-46) is an example of an advanced CSC that is provided with Real-Time Workshop Embedded Coder.

The following sections explain how to define advanced CSC types.

Create Your Own Parameter and Signal Classes

The first step is to use the Simulink Data Class Designer to create your own package containing classes derived from `Simulink.Parameter` or `Simulink.Signal`. This procedure is described in "Creating Packages with CSC Definitions" on page 3-38.

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

Create a Custom Attributes Class for Your CSC (Optional)

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

For example, the `ExportToFile` custom storage class requires that you set the `RTWInfo.CustomAttributes.HeaderFile` property to specify a `.h` file used for exporting each piece of data. See "Predefined CSCs" on page 3-7 for further information on instance-specific properties.

Write TLC Code for Your CSC

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

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

   ```
   @PackageName/tlc
   ```

2. Copy `TEMPLATE_v1.tlc` (or another CSC template) from `*matlabroot*/toolbox/rtw/targets/ecoder/csc_templates` into your `tlc` directory to use as a starting point for defining your custom storage class.

3. Write your TLC code, following the comments in the CSC template file. Comments describe how to specify code generation for data of your custom storage class (for example, how data structures are to be declared, defined, and whether they are accessed by value or by reference).

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

Register Custom Storage Class Definitions

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

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

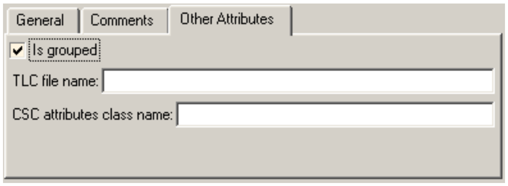
To register your class definitions:

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

```
cscdesigner -advanced
```

2 Select your package and create a new custom storage class.

3 Set the **Type** of the custom storage class to `Other`. Note that when you do this, the **Other Attributes** pane is displayed. This pane is visible only for CSCs whose **Type** is set to `Other`.



☑ Is grouped
TLC file name:
CSC attributes class name:

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

4 Set the properties shown on the **Other Attributes** pane. The properties are:

- **Is grouped**: Select this option if you intend to combine multiple data objects of this CSC into a single variable in the generated code. (for example, a struct).
- **TLC file name**: Enter the name of the TLC file corresponding to this custom storage class. The location of the file is assumed to be in the `/tlc` subdirectory for the package, so you should not enter the path to the file.
- **CSC attributes class name**: (optional) If you created a custom attributes class corresponding to this custom storage class, enter the full name of the custom attributes class. (see "Create a Custom Attributes Class for Your CSC (Optional)" on page 3-43).

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

GetSet Custom Storage Class for Data Store Memory

In this section...

Overview

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

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

The instance-specific properties of the `GetSet` storage class are summarized in GetSet Storage Class Properties on page 3-46.

GetSet Storage Class Properties

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

GetSet Storage Class Properties (Continued)

Property	Description
`SetFunction`	String that specifies the name of a function call to write data.
`HeaderFile` (optional)	String that specifies the name of a header (`.h`) file to add as an `#include` in the generated code. **Note** If you omit the `HeaderFile` property for a `GetSet` data object, you must specify a header file by an alternative means, such as the **Header file** field of the **Real-Time Workshop/Custom Code** pane of the Configuration Parameters dialog box. Otherwise, the generated code might not compile or might function improperly.

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

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

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

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

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

```
cscdesigner -advanced
```

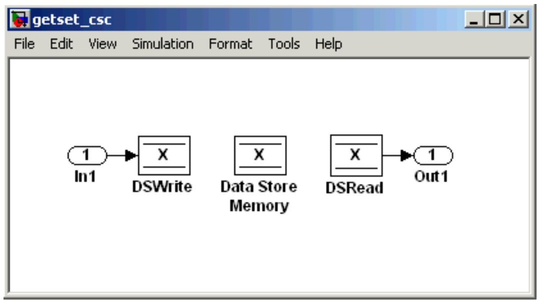
For more details about the definition of the `GetSet` storage class, look at its associated TLC code in the file

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

Example of Generated Code with GetSet Custom Storage Class

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

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



The following code is generated for this model:

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

   void getset_csc_step(void)
   {
     /* local block i/o variables */
     real_T rtb_DSRead_o;

     /* DataStoreWrite: '<Root>/DSWrite' incorporates:
      *   Inport: '<Root>/In1'
      */
     set_X(getset_csc_U.In1);

     /* DataStoreRead: '<Root>/DSRead' */
     rtb_DSRead_o = get_X();

     /* Outport: '<Root>/Out1' */
     getset_csc_Y.Out1 = rtb_DSRead_o;
   }
```

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

Setting Code Generation Options for Custom Storage Classes

The following code generation options affect the operation of CSCs:

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

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

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

Custom Storage Class Limitations

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

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

  - If data is assigned a grouped CSC, the CSC's **Data scope** property must be `Imported` and the data declaration must be provided in a user-supplied header file.
  - If data is assigned an ungrouped CSC (for example, `Const`) and the data's **Data scope** property is `Exported`, its **Header file** property must be unspecified. This results in the data being exported with the standard header file, *`model`*`.h`. Note that for ungrouped data, the **Data scope** and **Header file** properties are either specified by the selected CSC, or as one of the data object's instance-specific properties.

Older Custom Storage Classes (Prior to Release 14)

In this section...

Introduction

In releases prior to Real-Time Workshop Embedded Coder 4.0 (MATLAB Release 14), custom storage classes were implemented with special `Simulink.CustomSignal` and `Simulink.CustomParameter` classes. This section describes these older classes.

Note Models that use the `Simulink.CustomSignal` and `Simulink.CustomParameter` classes continue to operate correctly. The current CSCs support a superset of the functions of the older classes. Therefore, you should consider using the `Simulink.Signal` and `Simulink.Parameter` classes instead (see "Compatibility Issues for Older Custom Storage Classes" on page 3-60).

Simulink.CustomParameter Class

This class is a subclass of `Simulink.Parameter`. Objects of this class have expanded `RTWInfo` properties. The properties of `Simulink.CustomParameter` objects are:

- `RTWInfo.StorageClass`. This property should always be set to the default value, `Custom`.

- `RTWInfo.CustomStorageClass`. This property takes on one of the enumerated values described in the tables below. This property controls the generated storage declaration and code for the object.
- `RTWInfo.CustomAttributes`. This property defines additional attributes that are exclusive to the class, as described in “Instance-Specific Attributes for Older Storage Classes” on page 3-57.
- `Value`. This property is the numeric value of the object, used as an initial (or inlined) parameter value in generated code.

Simulink.CustomSignal Class

This class is a subclass of `Simulink.Signal`. Objects of this class have expanded `RTWInfo` properties. The properties of `Simulink.CustomSignal` objects are:

- `RTWInfo.StorageClass`. This property should always be set to the default value, `Custom`.
- `RTWInfo.CustomStorageClass`. This property takes on one of the enumerated values described in the tables below. This property controls the generated storage declaration and code for the object.
- `RTWInfo.CustomAttributes`. This optional property defines additional attributes that are exclusive to the storage class, as described in "Instance-Specific Attributes for Older Storage Classes" on page 3-57.

The following tables summarize the predefined custom storage classes for `Simulink.CustomSignal` and `Simulink.CustomParameter` objects. The entry for each class indicates

- Name and purpose of the class.
- Whether the class is valid for parameter or signal objects. For example, you can assign the storage class `Const` to a parameter object. This storage class is not valid for signals, however, since signal data (except for the case of invariant signals) is not constant.
- Whether the class is valid for complex data or nonscalar (wide) data.
- Data types supported by the class.

The first three classes, shown in Const, ConstVolatile, and Volatile Storage Classes (Prior to Release 14) on page 3-55, insert type qualifiers in the data declaration.

Const, ConstVolatile, and Volatile Storage Classes (Prior to Release 14)

Class Name	Purpose	Parameters	Signals	Data Types	Complex	Wide
`Const`	Use `const` type qualifier in declaration	Y	N	any	Y	Y
`ConstVolatile`	Use `const volatile` type qualifier in declaration	Y	N	any	Y	Y
`Volatile`	Use `volatile` type qualifier in declaration	Y	Y	any	Y	Y

The second set of three classes, shown in ExportToFile, ImportFromFile, and Internal Storage Classes (Prior to Release 14) on page 3-55, handles issues of data scope and file partitioning.

ExportToFile, ImportFromFile, and Internal Storage Classes (Prior to Release 14)

Class Name	Purpose	Parameters	Signals	Data Types	Complex	Wide
`ExportToFile`	Generate and include files, with user-specified name, containing global variable declarations and definitions	Y	Y	any	Y	Y

ExportToFile, ImportFromFile, and Internal Storage Classes (Prior to Release 14) (Continued)

Class Name	Purpose	Parameters	Signals	Data Types	Complex	Wide
`ImportFromFile`	Include predefined header files containing global variable declarations	Y	Y	any	Y	Y
`Internal`	Declare and define global variables whose scope is limited to the code generated by the Real-Time Workshop	Y	Y	any	Y	Y

The final three classes, shown in BitField, Define, and Struct Storage Classes (Prior to Release 14) on page 3-56, specify the data structure or construct used to represent the data.

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

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

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

Class Name	Purpose	Parameters	Signals	Data types	Complex	Wide
`Define`	Represent parameters with a `#define` macro	Y	N	any	N	N
`Struct`	Embed data in a named struct to encapsulate sets of data	Y	Y	any	N	Y

Instance-Specific Attributes for Older Storage Classes

Some custom storage classes have attributes that are exclusive to the class. These attributes are made visible as members of the `RTWInfo.CustomAttributes` field. For example, the `BitField` class has a `BitFieldName` attribute (`RTWInfo.CustomAttributes.BitFieldName`).

Additional Properties of Custom Storage Classes (Prior to Release 14) on page 3-58 summarizes the storage classes with additional attributes, and the meaning of those attributes. Attributes marked optional have default values and may be left unassigned.

Additional Properties of Custom Storage Classes (Prior to Release 14)

Storage Class Name	Additional Properties	Description	Optional (has default)
`ExportToFile`	`FileName`	String. Defines the name of the generated header file within which the global variable declaration should reside. If unspecified, the declaration is placed in *`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 (for example, `#include "vars.h"`) or angle brackets (for example, `#include <vars.h>`). The default is quotes.	Y
`BitField`	`BitFieldName`	String. Defines name of bit field in which data is embedded; if unassigned, the name defaults to `rt_BitField`.	Y
`Struct`	`StructName`	String. Defines name of the struct in which data is embedded; if unassigned, the name defaults to `rt_Struct`.	Y

Assigning a Custom Storage Class to Data

You can create custom parameter or signal objects from the MATLAB command line. For example, the following commands create a custom parameter object `p` and a custom signal object `s`:

```
p = Simulink.CustomParameter
s = Simulink.CustomSignal
```

After creating the object, set the `RTWInfo.CustomStorageClass` and `RTWInfo.CustomAttributes` fields. For example, the following commands sets these fields for the custom parameter object `p`:

```
p.RTWInfo.CustomStorageClass = 'ExportToFile'
p.RTWInfo.CustomAttributes.FileName = 'testfile.h'
```

Finally, make sure that the `RTWInfo.StorageClass` property is set to `Custom`. If you inadvertently set this property to some other value, the custom storage properties are ignored.

Code Generation with Older Custom Storage Classes

The procedure for generating code with data objects that have a custom storage class is similar to the procedure for code generation using Simulink data objects that have built-in storage classes. If you are unfamiliar with this procedure, see the discussion of Simulink data objects in the "Working with Data Structures" section of the Real-Time Workshop documentation.

To generate code with custom storage classes, you must

1. Create one or more data objects of class `Simulink.CustomParameter` or `Simulink.CustomSignal`.

2. Set the custom storage class property of the objects, as well as the class-specific attributes (if any) of the objects.

3. Reference these objects as block parameters, signals, block states, or Data Store memory.

When generating code from a model employing custom storage classes, make sure that the **Ignore custom storage classes** option is *not* selected. This is the default for Real-Time Workshop Embedded Coder.

When **Ignore custom storage classes** is selected:

- Objects with custom storage classes are treated as if their storage class attribute is set to `Auto`.
- The storage class of signals that have custom storage classes is not displayed on the signal line, even if the **Storage class** option of the Simulink **Format** menu is selected.

Ignore custom storage classes lets you switch to a target that does not support CSCs, such as the generic real-time target (GRT), without having to reconfigure your parameter and signal objects.

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

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

```
-aIgnoreCustomStorageClasses=1
```

Compatibility Issues for Older Custom Storage Classes

In Release 14, the full functionality of the `Simulink.CustomSignal` and `Simulink.CustomParameter` classes was added to the `Simulink.Signal` and `Simulink.Parameter` classes. You should consider replacing the use of `Simulink.CustomSignal` and `Simulink.CustomParameter` objects by using equivalent `Simulink.Signal` and `Simulink.Parameter` objects.

If you prefer, you can continue to use the `Simulink.CustomSignal` and `Simulink.CustomParameter` classes in the current release. Note that the following changes have been implemented in these classes:

- The `Internal` storage class has been removed from the enumerated values of the `RTWInfo.CustomStorageClass` property. `Internal` storage class is no longer supported.

- For the `ExportToFile` and `ImportFromFile` storage classes, the `RTWInfo.CustomAttributes.FileName` and `RTWInfo.CustomAttributes.IncludeDelimeter` properties have been obsoleted and combined into a single property, `RTWInfo.CustomAttributes.HeaderFile`. When specifying a header file, include both the filename and the required delimiter as you want them to appear in generated code, as in the following example:

  ```
  myobj.RTWInfo.CustomAttributes.HeaderFile = '<myheader.h>';
  ```

- Prior to Release 14, user-defined CSCs were created by designing custom packages that included the CSC definitions. This technique for creating CSCs is obsolete; see “Creating Packages with CSC Definitions” on page 3-38 for a description of the current procedure, which is much simpler.

  If you designed your own custom packages containing CSCs prior to Release 14 you should convert them to Release 14 CSCs. The conversion procedure is described in the next section, “Converting Older Packages to Use CSC Registration Files” on page 3-61.

Converting Older Packages to Use CSC Registration Files

A Simulink data class package can be associated with one or more CSC definitions. In Release 14, the linkage between a set of CSC definitions and a package is formed when a CSC registration file (`csc_registration.m`) is located in the package directory.

Prior to Release 14, user-defined CSCs were created by designing custom packages that included the CSC definitions as part of the package.

The Simulink Data Class Designer supports conversion of older packages to the use of CSC registration files. When such a package is selected in Simulink Data Class Designer, a special conversion button is displayed on the **Custom Storage Classes** pane. This button lets you invoke a conversion procedure; you can then write out all files and directories required to define the package, including a CSC registration file. To convert a package:

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

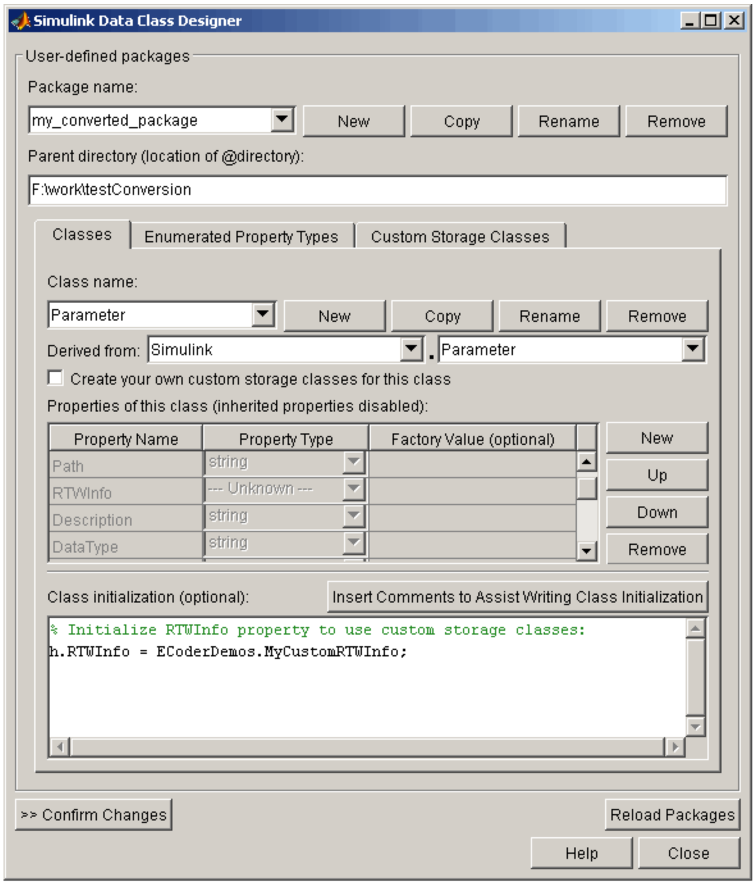
2 Open the Simulink Data Class Designer by typing the following command at the MATLAB command prompt:

```
sldataclassdesigner
```

3 The Data Class Designer loads all packages that exist on the MATLAB path. Select the package to be converted from the **Package name** menu. Then, click **OK**.

4 If you want to store the converted package in a different directory than the original package, enter the desired path in the **Parent directory** field. This step is optional.

   The figure below shows the package `my_converted_package`. The package definition is stored in `d:\work\testConversion`.


Simulink Data Class Designer
User-defined packages
Package name:
New
Copy
Rename
Remove
Classes
Enumerated Property Types
Custom Storage Classes
Class name:
New
Copy
Rename
Remove
Derived from:
Create your own custom storage classes for this class
Properties of this class (inherited properties disabled):
New
Up
Down
Remove
Class initialization (optional):
Insert Comments to Assist Writing Class Initialization
>> Confirm Changes
Reload Packages
Help
Close

Package name: my_converted_package

Parent directory (location of @directory): F:\work\testConversion

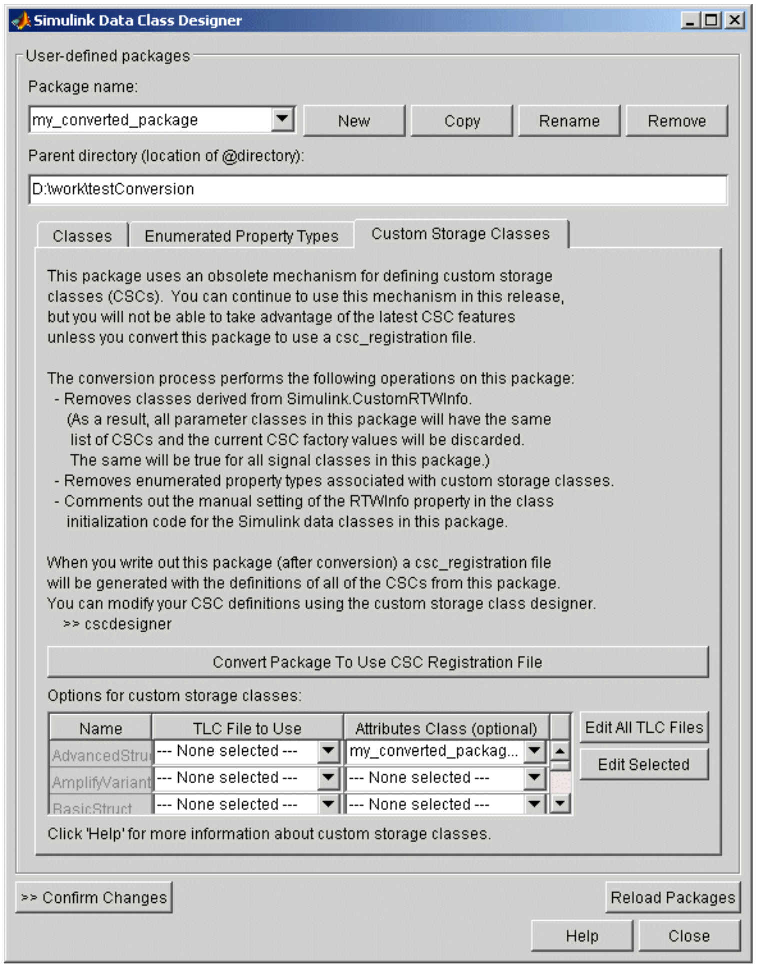
Class name: Parameter

Derived from: Simulink . Parameter

Property Name	Property Type	Factory Value (optional)
Path	string	
RTWInfo	--- Unknown ---	
Description	string	
DataType	string	

```
% Initialize RTWInfo property to use custom storage classes:
h.RTWInfo = ECoderDemos.MyCustomRTWInfo;
```

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



Package name: my_converted_package

Parent directory (location of @directory): D:\work\testConversion

This package uses an obsolete mechanism for defining custom storage classes (CSCs). You can continue to use this mechanism in this release, but you will not be able to take advantage of the latest CSC features unless you convert this package to use a csc_registration file.

The conversion process performs the following operations on this package:
- Removes classes derived from Simulink.CustomRTWInfo.
  (As a result, all parameter classes in this package will have the same list of CSCs and the current CSC factory values will be discarded. The same will be true for all signal classes in this package.)
- Removes enumerated property types associated with custom storage classes.
- Comments out the manual setting of the RTWInfo property in the class initialization code for the Simulink data classes in this package.

When you write out this package (after conversion) a csc_registration file will be generated with the definitions of all of the CSCs from this package. You can modify your CSC definitions using the custom storage class designer.

```
   >> cscdesigner
```

Options for custom storage classes:

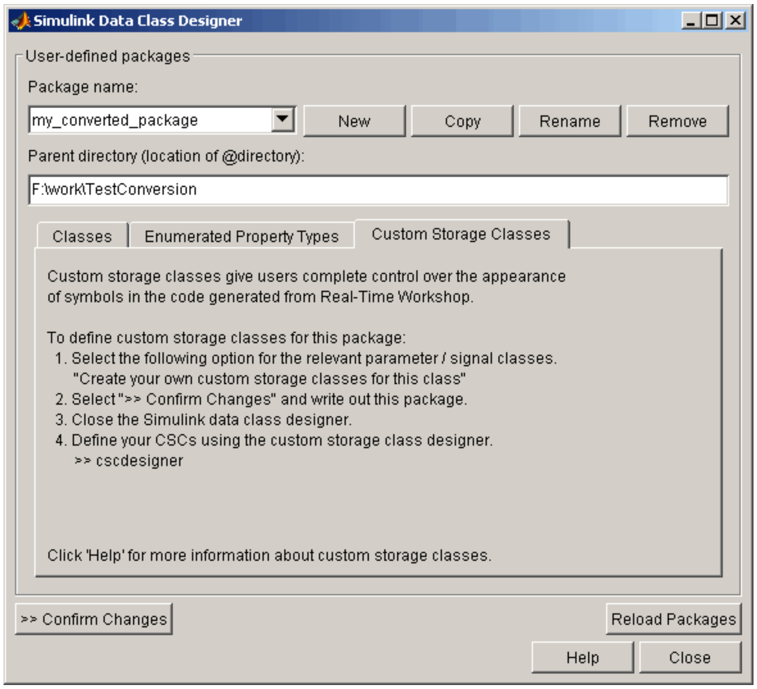
Name	TLC File to Use	Attributes Class (optional)
AdvancedStru	--- None selected ---	my_converted_packag...
AmplifyVariant	--- None selected ---	--- None selected ---
BasicStruct	--- None selected ---	--- None selected ---

Click 'Help' for more information about custom storage classes.

Below the message text, the pane also contains a button captioned **Convert Package to Use CSC Registration File**. This button invokes a script that converts the package to use a CSC registration file.

Note that this button does not actually create the CSC registration file. That happens when the package files are written out, as described below.

6 Click **Convert Package to Use CSC Registration File**. After conversion, the appearance of the pane changes, as shown below.



Custom storage classes give users complete control over the appearance of symbols in the code generated from Real-Time Workshop.

To define custom storage classes for this package:
1. Select the following option for the relevant parameter / signal classes. "Create your own custom storage classes for this class"
2. Select ">> Confirm Changes" and write out this package.
3. Close the Simulink data class designer.
4. Define your CSCs using the custom storage class designer. >> cscdesigner

Click 'Help' for more information about custom storage classes.

7 Click **Confirm Changes**. In the **Confirm Changes** pane, select the package you converted. Add the parent directory to the MATLAB path if necessary. Then, click **Write Selected**.

8 Click **Close**.

9 You can now view and edit the CSCs belonging to your package in the Custom Storage Class Designer. To do so, type the following command at the MATLAB prompt:

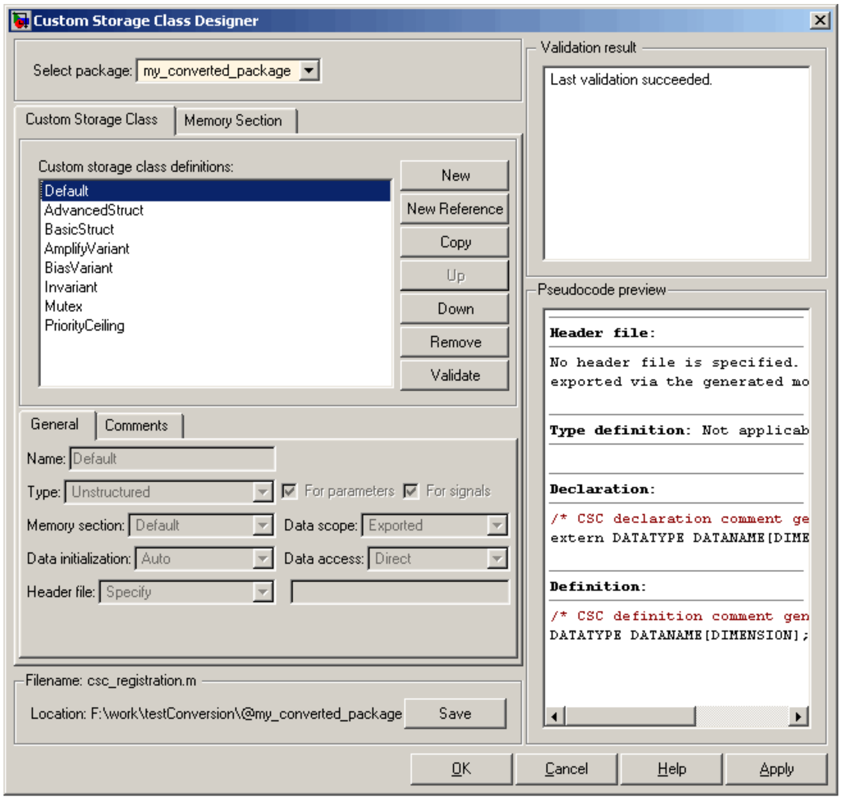
```
cscdesigner
```

Note You must launch the CSC Designer with the `-advanced` motion to edit the attributes of old CSCs because they are defined with user-defined TLC files.

The Custom Storage Class Designer loads all packages that have a CSC registration file.

10 Select your converted package from the **Select package** menu.

The figure below shows the Custom Storage Class Designer displaying the CSCs defined in the package `my_converted_package`. See "Designing Custom Storage Classes and Memory Sections" on page 3-16 for a description of the operation of the Custom Storage Class Designer.



Select package: my_converted_package

Custom storage class definitions:
Default
AdvancedStruct
BasicStruct
AmplifyVariant
BiasVariant
Invariant
Mutex
PriorityCeiling

Name: Default
Type: Unstructured ✓ For parameters ✓ For signals
Memory section: Default Data scope: Exported
Data initialization: Auto Data access: Direct
Header file: Specify

Filename: csc_registration.m
Location: F:\work\testConversion\@my_converted_package

Validation result
Last validation succeeded.

Pseudocode preview

```
Header file:

No header file is specified.
exported via the generated mo

Type definition: Not applicab

Declaration:

/* CSC declaration comment ge
extern DATATYPE DATANAME[DIME

Definition:

/* CSC definition comment gen
DATATYPE DATANAME[DIMENSION];
```

Note All user-defined CSCs created prior to Release 14 are defined with their own TLC code. As a result, after conversion, the **Type** is set to `Other` (as opposed to `Unstructured` or `FlatStructure`). See “Defining Advanced Custom Storage Class Types” on page 3-42 for more information.

---

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

Memory Sections

Introduction to Memory Sections

In this section...

Overview

Real-Time Workshop Embedded Coder provides a memory section capability that allows you to insert comments and pragmas into the generated code for

- Data in custom storage classes
- Model-level functions
- Model-level internal data
- Subsystem functions
- Subsystem internal data

Pragmas inserted into generated code can surround

- A contiguous block of function or data definitions
- Each function or data definition separately

When pragmas surround each function or data definition separately, the text of each pragma can contain the name of the definition to which it applies.

Memory Sections Demo

To see a demo of memory sections, type `rtwdemo_memsec` in the MATLAB Command Window.

Additional Information

See the following for additional information relating to memory sections:

- Simulink data types, packages, data classes, and data objects:
  - “Working with Data” in the Simulink documentation
- Real-Time Workshop data structures and storage classes:
  - “Working with Data Structures” in the Real-Time Workshop documentation
- Real-Time Workshop Embedded Coder custom storage classes:
  - Chapter 3, “Custom Storage Classes” in the Real-Time Workshop Embedded Coder documentation
- Fine-tuned optimization of generated code for functions or data:
  - The *Real-Time Workshop Target Language Compiler* documentation

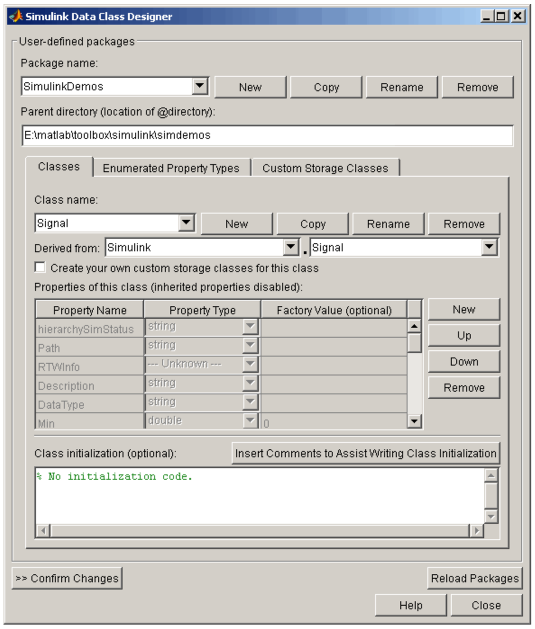
Requirements for Defining Memory Sections

Before you can define memory sections, you must do the following:

1 Set the Simulink model's code generation target to an embedded target such as `ert.tlc`.

2 If you need to create packages, specify package properties, or create classes, including custom storage classes, choose **Tools > Data Class Designer** in the model window.

A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the Simulink Data Class Designer appears:



Package name: SimulinkDemos

Parent directory (location of @directory): E:\matlab\toolbox\simulink\simdemos

Class name: Signal

Derived from: Simulink . Signal

☐ Create your own custom storage classes for this class

Properties of this class (inherited properties disabled):

Property Name	Property Type	Factory Value (optional)
hierarchySimStatus	string	
Path	string	
RTWInfo	--- Unknown ---	
Description	string	
DataType	string	
Min	double	0

Class initialization (optional):

```
% No initialization code.
```

Instructions for using the Simulink Data Class Designer appear in "Working with Data" in the Simulink documentation. See also the instructions that appear when you click the **Custom Storage Classes** tab.

3 If you need to specify custom storage class properties,

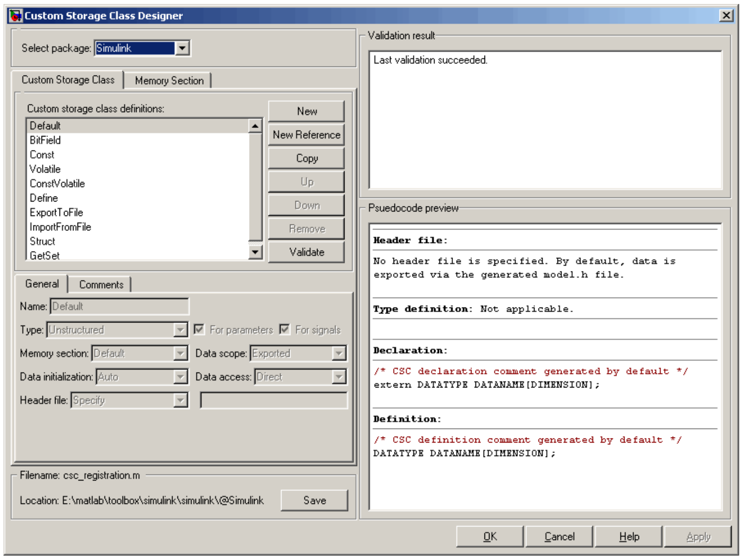
a Choose **View > Model Explorer** in the model window.

The Model Explorer appears.

b Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

c Select the **Custom Storage Class** tab. The **Custom Storage Class** pane looks like this:



Select package: Simulink

Custom storage class definitions:
Default
BitField
Const
Volatile
ConstVolatile
Define
ExportToFile
ImportFromFile
Struct
GetSet

Name: Default
Type: Unstructured ✓ For parameters ✓ For signals
Memory section: Default Data scope: Exported
Data initialization: Auto Data access: Direct
Header file: Specify

Filename: csc_registration.m
Location: E:\matlab\toolbox\simulink\simulink\@Simulink

Validation result
Last validation succeeded.

Psuedocode preview

```
Header file:
No header file is specified. By default, data is
exported via the generated model.h file.

Type definition: Not applicable.

Declaration:
/* CSC declaration comment generated by default */
extern DATATYPE DATANAME[DIMENSION];

Definition:
/* CSC definition comment generated by default */
DATATYPE DATANAME[DIMENSION];
```

d Use the **Custom Storage Class** pane as needed to specify custom storage class properties. Instructions for using this pane are in "Designing Custom Storage Classes and Memory Sections" on page 3-16.

Defining Memory Sections

In this section...

Editing Memory Section Properties

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

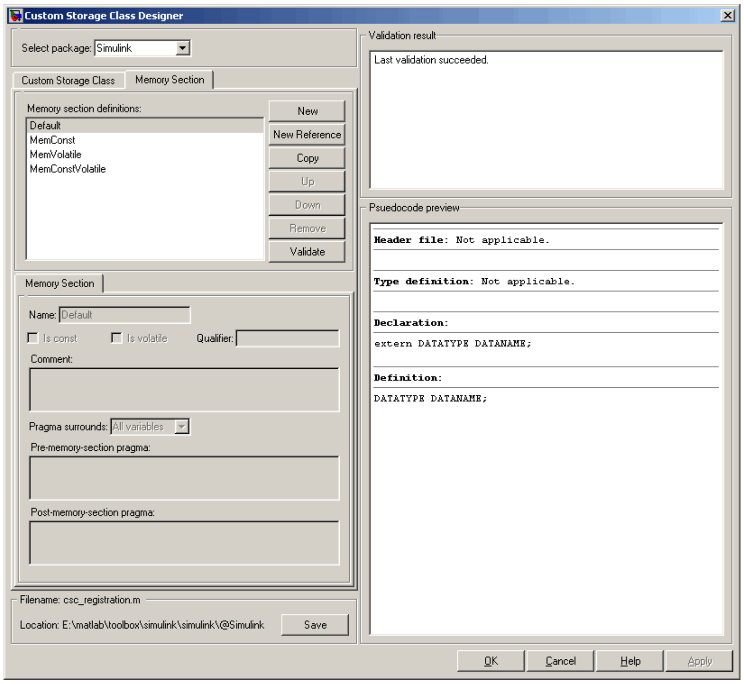
1. Choose **View > Model Explorer** in the model window.

   The Model Explorer appears.

2. Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

   A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

3. Click the **Memory Section** tab of the Custom Storage Class Designer. The **Memory Section** pane looks like this:



Last validation succeeded.

```
Header file: Not applicable.

Type definition: Not applicable.

Declaration:
extern DATATYPE DATANAME;

Definition:
DATATYPE DATANAME;
```

The rest of this section describes the use of the **Memory section** subpane on the lower left. For descriptions of the other subpanes, instructions for validating memory section definitions, and other information, see "Editing Memory Section Definitions" on page 3-31.

Specifying the Memory Section Name

To specify the name of a memory section, use the **Name** field. A memory section name must be a legal MATLAB identifier.

Specifying a Qualifier for Custom Storage Class Data Definitions

To specify a qualifier for custom storage class data definitions in a memory section, enter the components of the qualifier below the **Name** field.

- To specify `const`, check **Is const**.
- To specify `volatile`, check **Is volatile**.
- To specify anything else (e.g., `static`), enter the text in the **Qualifier** field.

The qualifier will appear in generated code with its components in the same left-to-right order in which their definitions appear in the dialog box. A preview appears in the **Pseudocode preview** subpane on the lower right.

Note Specifying a qualifier affects only custom storage class data definitions. The code generator omits the qualifier from any other category of definition.

Specifying Comment and Pragma Text

To specify a comment, pre-pragma, or post-pragma for a memory section, enter the text in the appropriate edit boxes on the left side of the Custom Storage Class Designer. These boxes accept multiple lines separated by ordinary Returns.

Surrounding Individual Definitions with Pragmas

If the **Pragma surrounds** field for a memory section specifies `Each variable`, the code generator will surround each definition in a contiguous block of definitions with the comment, pre-pragma, and post-pragma defined for the section. This behavior occurs with all categories of definitions.

If the **Pragma surrounds** field for a memory section specifies `All variables`, the code generator will insert the comment and pre-pragma for the section before the first definition in a contiguous block of custom storage class data definitions, and the post-pragma after the last definition in the block.

Note Specifying `All variables` affects only custom storage class data definitions. For any other category of definition, the code generator surrounds each definition separately regardless of the value of **Pragma surrounds**.

Including Identifier Names in Pragmas

When pragmas surround each separate definition in a contiguous block, you can include the string `%<identifier>` in a pragma. The string must appear without surrounding quotes.

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

You can use `%<identifier>` with pragmas *only* when pragmas to surround each variable. The `Validate` phase will report an error if you violate this rule.

Note Although `%<identifier>` looks like a TLC variable, it is not: it is just a keyword that directs the code generator to substitute the applicable data definition identifier when it outputs a pragma. TLC variables cannot appear in pragma specifications in the **Memory Section** pane.

Applying Memory Sections

In this section...

Assigning Memory Sections to Custom Storage Classes

To assign a memory section to a custom storage class,

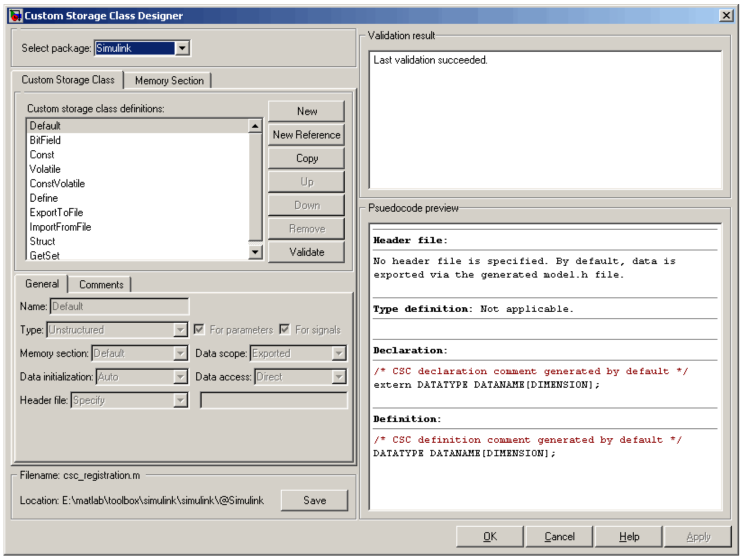
1 Choose **View > Model Explorer** in the model window.

   The Model Explorer appears.

2 Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

   A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

3 Select the **Custom Storage Class** tab. The **Custom Storage Class** pane looks like this:



Select package: Simulink

Custom storage class definitions:

Default
BitField
Const
Volatile
ConstVolatile
Define
ExportToFile
ImportFromFile
Struct
GetSet

Name: Default

Type: Unstructured ✓ For parameters ✓ For signals

Memory section: Default Data scope: Exported

Data initialization: Auto Data access: Direct

Header file: Specify

Filename: csc_registration.m

Location: E:\matlab\toolbox\simulink\simulink\@Simulink

Validation result

Last validation succeeded.

Psuedocode preview

```
Header file:
No header file is specified. By default, data is
exported via the generated model.h file.

Type definition: Not applicable.

Declaration:
/* CSC declaration comment generated by default */
extern DATATYPE DATANAME[DIMENSION];

Definition:
/* CSC definition comment generated by default */
DATATYPE DATANAME[DIMENSION];
```

4. Select the desired custom storage class in the **Custom storage class definitions** pane.

5. Select the desired memory section from the **Memory section** pull-down.

6. Click **Apply** to apply changes to the open copy of the model; **Save** to apply changes and save them to disk; or **OK** to apply changes, save changes, and close the Custom Storage Class Designer.

Generated code for all data definitions in the specified custom storage class will be enclosed in the pragmas of the specified memory section. The pragmas can surround contiguous blocks of definitions or each definition separately, as described in "Surrounding Individual Definitions with Pragmas" on page 4-9. For more information, see "Creating Packages with CSC Definitions" on page 3-38.

Applying Memory Sections to Model-Level Functions and Internal Data

When using Real-Time Workshop Embedded Coder, you can apply memory sections to the following categories of model-level functions:

Function Category	**Function Subcategory**
Initialize/Terminate functions	Initialize/Start
	Terminate
Execution functions	Step functions
	Run-time initialization
	Derivative
	Enable
	Disable

When using Real-Time Workshop Embedded Coder, you can apply memory sections to the following categories of internal data:

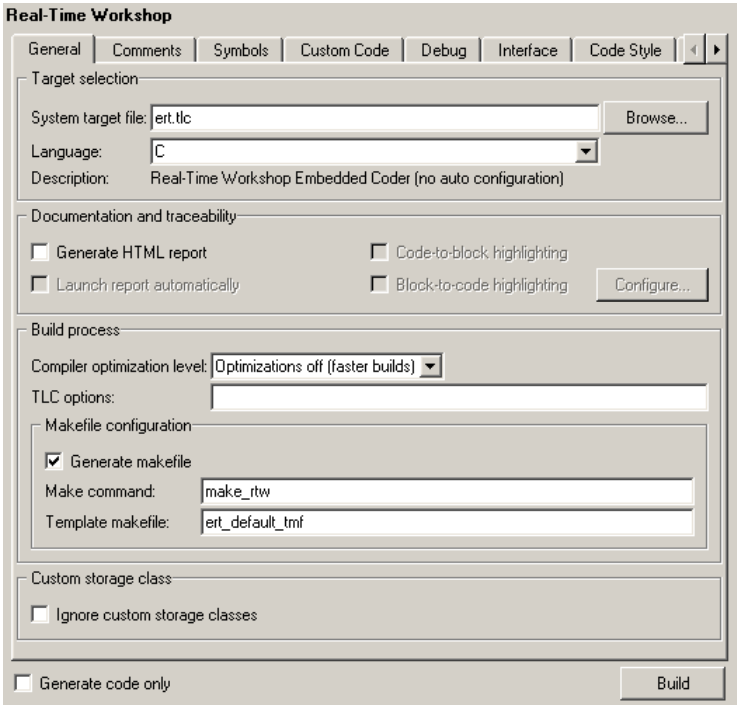
Data Category	**Data Definition**	**Data Purpose**
Constants	`model_cP`	Constant parameters
	`model_cB`	Constant block I/O
	`model_Z`	Zero representation
Input/Output	`model_U`	Root inputs
	`model_Y`	Root outputs
Internal data	`model_B`	Block I/O
	`model_D`	D-work vectors
	`model_M`	Run-time model
	`model_Zero`	Zero-crossings
Parameters	`model_P`	Parameters

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems *except* atomic subsystems that contain overriding memory section specifications, as described in “Applying Memory Sections to Atomic Subsystems” on page 4-15.

To specify memory sections for model-level functions or internal data,

1 Open the Model Explorer and select **Configuration (Active)** > **Real-Time Workshop** > **General**. (Alternatively, choose **Simulation > Configuration Parameters** in the model window.)

2 Ensure that the **System target file** is an ERT target, such as `ert.tlc` .



Real-Time Workshop

General | Comments | Symbols | Custom Code | Debug | Interface | Code Style

Target selection

System target file: ert.tlc

Language: C

Description: Real-Time Workshop Embedded Coder (no auto configuration)

Documentation and traceability

☐ Generate HTML report
☐ Launch report automatically
☐ Code-to-block highlighting
☐ Block-to-code highlighting

Build process

Compiler optimization level: Optimizations off (faster builds)

TLC options:

Makefile configuration

☑ Generate makefile

Make command: make_rtw

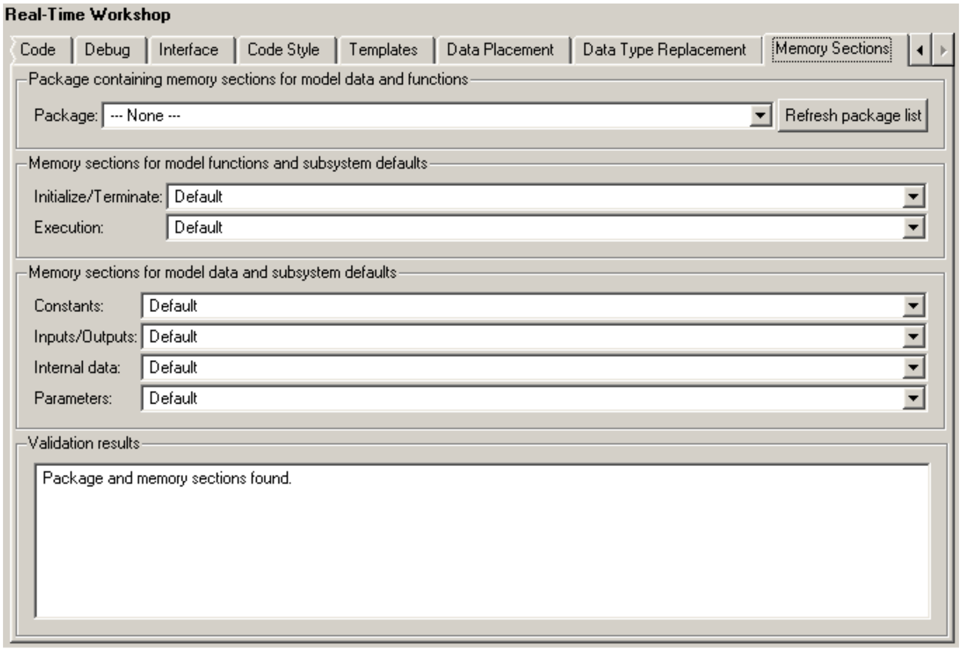
Template makefile: ert_default_tmf

Custom storage class

☐ Ignore custom storage classes

☐ Generate code only

3 Select the **Memory Sections** tab. The **Memory Sections** pane looks like this:



Package containing memory sections for model data and functions

Package: --- None ---

Memory sections for model functions and subsystem defaults

Initialize/Terminate: Default

Execution: Default

Memory sections for model data and subsystem defaults

Constants: Default

Inputs/Outputs: Default

Internal data: Default

Parameters: Default

Validation results

Package and memory sections found.

4 Initially, the **Package** field specifies `---None---` and the pull-down lists only built-in packages. If you have defined any packages of your own, click **Refresh package list**. This action adds all user-defined packages on your search path to the package list.

5 In the **Package** pull-down, select the package that contains the memory sections that you want to apply.

6 In the pull-down for each category of internal data and model-level function, specify the memory section (if any) that you want to apply to that category. Accepting or specifying `Default` omits specifying any memory section for that category.

7 Click **Apply** to save any changes to the package and memory section selections.

Applying Memory Sections to Atomic Subsystems

For any atomic subsystem whose generated code format is `Function` or `Reusable Function`, you can specify memory sections for functions and internal data that exist in that code format. Such specifications override any model-level memory section specifications. Such overrides apply only to the atomic subsystem itself, not to any subsystems within it. Subsystems of an atomic subsystem inherit memory section specifications from the top-level model, *not* from the containing atomic subsystem.

To specify memory sections for an atomic subsystem,

1. Right-click the subsystem in the model window.

2. Choose **Subsystem Parameters** from the context menu. The Function Block Parameters: *`Subsystem`* dialog box appears.

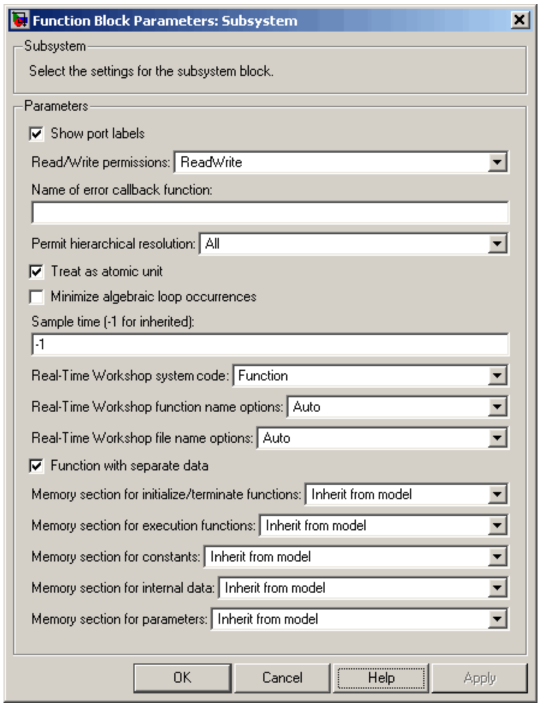
3. Ensure that **Treat as atomic unit** is checked. Otherwise, you cannot specify memory sections for the subsystem.

   For an atomic system, you can use the **Real-Time Workshop system code** field to control the format of the generated code.

4. Ensure that **Real-Time Workshop system code** is `Function` or `Reusable function`. Otherwise, you cannot specify memory sections for the subsystem.

5. If the code format is `Function` and you want separate data, check **Function with separate data**.

The **Real-Time Workshop** pane now shows all applicable memory section options. The available options depend on the values of **Real-Time Workshop system code** and the **Function with separate data** check box. When the former is `Function` and the latter is checked, the pane looks like this:



Function Block Parameters: Subsystem

Subsystem

Select the settings for the subsystem block.

Parameters

- [✓] Show port labels
- Read/Write permissions: ReadWrite
- Name of error callback function:
- Permit hierarchical resolution: All
- [✓] Treat as atomic unit
- [ ] Minimize algebraic loop occurrences
- Sample time (-1 for inherited): -1
- Real-Time Workshop system code: Function
- Real-Time Workshop function name options: Auto
- Real-Time Workshop file name options: Auto
- [✓] Function with separate data
- Memory section for initialize/terminate functions: Inherit from model
- Memory section for execution functions: Inherit from model
- Memory section for constants: Inherit from model
- Memory section for internal data: Inherit from model
- Memory section for parameters: Inherit from model

6 In the pull-down for each available definition category, specify the memory section (if any) that you want to apply to that category.

- Selecting `Inherit from model` inherits the corresponding selection (if any) from the model level (not any parent subsystem).
- Selecting `Default` specifies that the category has no associated memory section, overriding any model-level specification for that category.

7 Click **Apply** to save changes, or **OK** to save changes and close the dialog box.

Caution If you use **Build Subsystem** to generate code for an atomic subsystem that specifies memory sections, the code generator ignores the subsystem-level specifications and uses the model-level specifications instead. The generated code is the same as if the atomic subsystem specified `Inherit from model` for every category of definition. For information about **Build Subsystem**, see “Generating Code and Executables from Subsystems”.

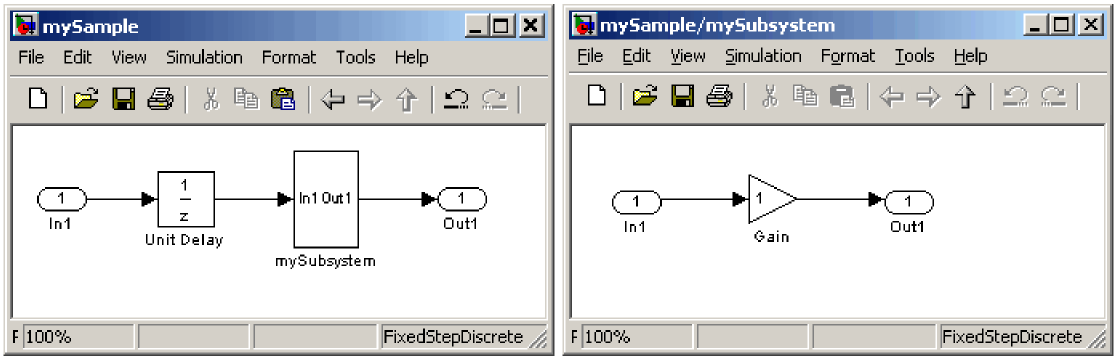
It is not possible to specify the memory section for a subsystem in a library. However, you can specify the memory section for the subsystem after you have copied it into a Simulink model. This is because in the library it is unknown what code generation target will be used. You can copy a library block into many different models with different code generation targets and different memory sections available.

Examples of Generated Code with Memory Sections

In this section...

Sample ERT-Based Model with Subsystem

The next figure shows an ERT-based Simulink model that defines one subsystem, and the contents of that subsystem.



Assume that the subsystem is atomic, the **Real-Time Workshop system code** is `Reusable function`, memory sections have been created and assigned as shown in the next two tables, and all data memory sections specify **Pragma surrounds** to be `Each variable`.

Model-Level Memory Section Assignments and Definitions

Section Assignment	Section Name	Field Name	Field Value
Input/Output	`MemSect1`	Pre-pragma	`#pragma IO_begin`
		Post-pragma	`#pragma IO-end`
Internal data	`MemSect2`	Pre-pragma	`#pragma InData-begin(%<identifier>)`
		Post-pragma	`#pragma InData-end`
Parameters	`MemSect3`	Pre-pragma	`#pragma Parameters-begin`
		Post-pragma	`#pragma Parameters-end`
Initialize/ Terminate	`MemSect4`	Pre-pragma	`#pragma InitTerminate-begin`
		Post-pragma	`#pragma InitTerminate-end`
Execution functions	`MemSect5`	Pre-pragma	`#pragma ExecFunc-begin(%<identifier>)`
		Post-pragma	`#pragma ExecFunc-begin(%<identifier>)`

Subsystem-Level Memory Section Assignments and Definitions

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

Given the preceding specifications and definitions, the code generator would create the following code, with minor variations depending on the current version of the Target Language Compiler.

Model-Level Data Structures

```
#pragma IO-begin
ExternalInputs_mySample mySample_U;
#pragma IO-end

#pragma IO-begin
```

```
ExternalOutputs_mySample mySample_Y;
#pragma IO-end

#pragma InData-begin(mySample_B)
BlockIO_mySample mySample_B;
#pragma InData-end

#pragma InData-begin(mySample_DWork)
D_Work_mySample mySample_DWork;
#pragma InData-end

#pragma InData-begin(mySample_M_)
RT_MODEL_mySample mySample_M_;
#pragma InData-end

#pragma InData-begin(mySample_M)
RT_MODEL_mySample *mySample_M = &mySample_M_;
#pragma InData-end

#pragma Parameters-begin
Parameters_mySample mySample_P = {
  0.0 , {2.3}
};
#pragma Parameters-end
```

Model-Level Functions

```
#pragma ExecFunc-begin(mySample_step)
void mySample_step(void)
{
  real_T rtb_UnitDelay;
  rtb_UnitDelay = mySample_DWork.UnitDelay_DSTATE;
  mySubsystem(rtb_UnitDelay, &mySample_B.mySubsystem;,
   (rtP_mySubsystem *) &mySample_P.mySubsystem);
  mySample_Y.Out1_o = mySample_B.mySubsystem.Gain;
  mySample_DWork.UnitDelay_DSTATE = mySample_U.In1;
}
#pragma ExecFunc-end(mySample_step)

#pragma InitTerminate-begin
void mySample_initialize(void)
{
  rtmSetErrorStatus(mySample_M, (const char_T *)0);
  {
    ((real_T*)&mySample_B.mySubsystem.Gain)[0] = 0.0;
  }
  mySample_DWork.UnitDelay_DSTATE = 0.0;
  mySample_U.In1 = 0.0;
  mySample_Y.Out1_o = 0.0;
  mySample_DWork.UnitDelay_DSTATE = mySample_P.UnitDelay_X0;
}
#pragma InitTerminate-end
```

Subsystem Function

Because the subsystem specifies a memory section for execution functions that overrides that of the parent model, subsystem code looks like this:

```
/* File: mySubsystem.c */

#pragma DATA_SEC(mySubsystem,  FAST_RAM )
void mySubsystem(real_T rtu_In1,
rtB_mySubsystem *localB,
rtP_mySubsystem *localP)
{
  localB->Gain = rtu_In1 * localP->Gain_Gain;
}
```

If the subsystem had not defined its own memory section for execution functions, but inherited that of the parent model, the subsystem code would have looked like this:

```
/* File: mySubsystem.c */

#pragma ExecFunc-begin(mySubsystem)
void mySubsystem(real_T rtu_In1,
rtB_mySubsystem *localB,
rtP_mySubsystem *localP)
{
  localB->Gain = rtu_In1 * localP->Gain_Gain;
}
#pragma ExecFunc-end(mySubsystem)
```

Advanced Code Generation Techniques

Introduction

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

- *User-defined data types*: How to use `Simulink.NumericType`, `Simulink.StructType` and other data type objects to map your own data type definitions to Simulink built-in data types.
- *Model configuration*: Several sections describe features that support automatic (as opposed to manual) configuration of model options for code generation. The information in each of these sections builds upon the previous section.
  - "Customizing the Target Build Process with the STF_make_rtw Hook File" on page 5-9 describes the general mechanism for adding target-specific customizations to the build process.
  - "Auto-Configuring Models for Code Generation" on page 5-22 shows how to use this mechanism (along with supporting utilities) to set model options affecting code generation automatically.
  - A similar mechanism is used by two special versions of the ERT target, optimized for fixed-point and floating-point code generation. These are described in "Generating Efficient Code with Optimized ERT Targets" on page 5-26.
  - "Optimizing Your Model with Configuration Wizard Blocks and Scripts" on page 5-61 describes a simpler approach to automatic model configuration. A library of Configuration Wizard blocks and scripts is provided to let you configure models quickly for common scenarios; you can also create your own scripts with minimal M-file programming.
- *Custom code generation*: These features let you directly customize generated code by creating template files that are invoked during the TLC code generation process. Basic knowledge of TLC is required to use these features.
  - "Custom File Processing" on page 5-34 describes a flexible and powerful TLC API that lets you emit custom code to any generated file (including both the standard generated model files and separate code modules).

  - "Generating Custom File Banners" on page 5-55 describes a simple way to generate file banners (useful for inserting your organization's copyrights and other common information into generated files).

- *Backward compatibility issues:* Read "Optimizing Your Model with Configuration Wizard Blocks and Scripts" on page 5-61 if you have created an `STF_rtw_info_hook` file for use with a custom target, prior to MATLAB Release 14. The `STF_rtw_info_hook` hook file mechanism for specifying target-specific characteristics for code generation has been supplanted by the simpler and more powerful **Hardware Implementation** pane of the Configuration Parameters dialog box.

Code Generation with User-Defined Data Types

In this section...

Overview

Real-Time Workshop Embedded Coder supports use of user-defined data type objects in code generation. These include objects of the following classes:

- `Simulink.AliasType`
- `Simulink.Bus`
- `Simulink.NumericType`
- `Simulink.StructType`

For information on the properties and usage of these data object classes, see `Simulink.AliasType`, `Simulink.Bus`, `Simulink.NumericType`, and `Simulink.StructType` in the "Data Object Classes" section of the Simulink Reference documentation. For general information on creating and using data objects, see the "Working with Data Objects" section of the Simulink documentation

In code generation, you can use user-defined data objects to

- Map your own data type definitions to Simulink built-in data types, and specify that your data types are to be used in generated code.
- Optionally, generate `#include` directives specifying your own header files, containing your data type definitions. This technique lets you use legacy data types in generated code.

In general, code generated from user-defined data objects conforms to the properties and attributes of the objects as defined for use in simulation. When generating code from user-defined data objects, the name of the object is the name of the data type that is used in the generated code. Exception: for `Simulink.NumericType` objects whose `IsAlias` property is false, the name of the functionally equivalent built-in or fixed-point Simulink data type is used instead.

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

Specifying Type Definition Location for User-Defined Data Types

When a model uses `Simulink.DataType` and `Simulink.Bus` objects, corresponding `typedefs` are needed in code. Both `Simulink.DataType` and `Simulink.Bus` objects have a `HeaderFile` property that controls the location of the object's `typedef`. Setting a `HeaderFile` is optional and affects code generation only.

Omitting a HeaderFile Value

If the `HeaderFile` property for a `Simulink.DataType` or `Simulink.Bus` object is left empty, a generated `typedef` for the object appears in the generated file *`model`*`_types.h`.

Example. For a `Simulink.NumericType` object named `myfloat` with a `Category` of `double` and no `HeaderFile` property specified, *`model`*`_types.h` in the generated code contains:

```
typedef real_T myfloat;
```

Specifying a HeaderFile Value

If the `HeaderFile` property for a `Simulink.DataType` or `Simulink.Bus` object is set to a string value,

- The string must be the name of a header file that contains a `typedef` for the object.

- The generated file *model*_types.h contains a #include that gives the header file name.

You can use this technique to include legacy or other externally created typedefs in generated code. When the generated code compiles, the specified header file must be accessible on the build process include path.

HeaderFile Property Syntax. The `HeaderFile` property should include the desired preprocessor delimiter (`""` or `'<>'`), as in the following examples.

This example:

```
myfloat.HeaderFile = '<legacy_types.h>'
```

generates the directive:

```
#include <legacy_types.h>
```

This example:

```
myfloat.HeaderFile = '"legacy_types.h>"'
```

generates the directive:

```
#include "legacy_types.h"
```

Using User-Defined Data Types for Code Generation

To specify and use user-defined data types for code generation:

1. Create a user-defined data object and configure its properties, as described in the “Working with Data Objects” section of the Simulink documentation.

2. If you specified the `HeaderFile` property, copy the header file to the appropriate directory.

3. Set the output data type of selected blocks in your model to the user-defined data object. To do this, set the **Data type** parameter of the block to `Specify with dialog`. Then, enter the object name in the **Output data type** parameter.

4. The specified output data type propagates through the model and variables of the user-defined type are declared as required in the generated code.

Customizing the Target Build Process with the STF_make_rtw Hook File

In this section...

Overview

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

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

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

File and Function Naming Conventions

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

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

STF`_make_rtw_hook.m` file to `mytarget_make_rtw_hook.m`. Likewise, the hook function implemented within the file should follow the same naming convention.

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

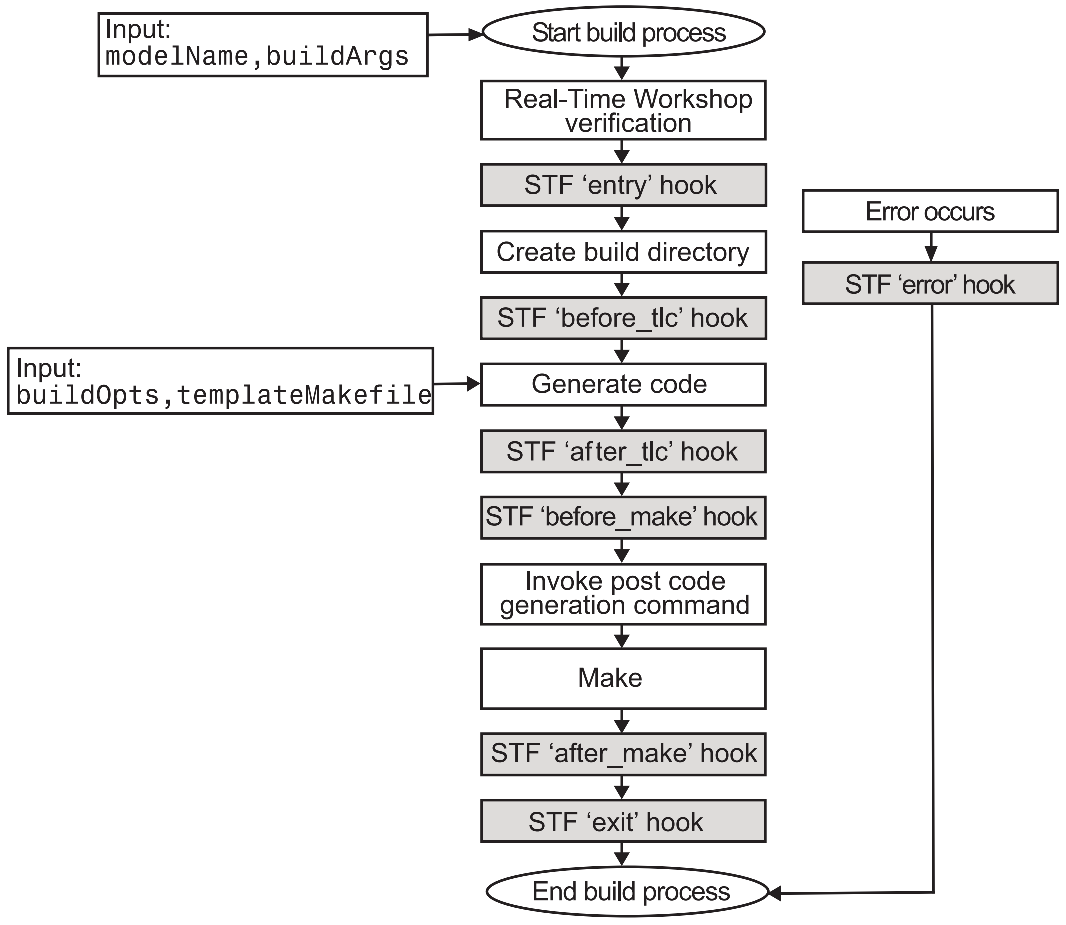
STF_make_rtw_hook.m Function Prototype and Arguments

The function prototype for *STF*_make_rtw_hook is

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

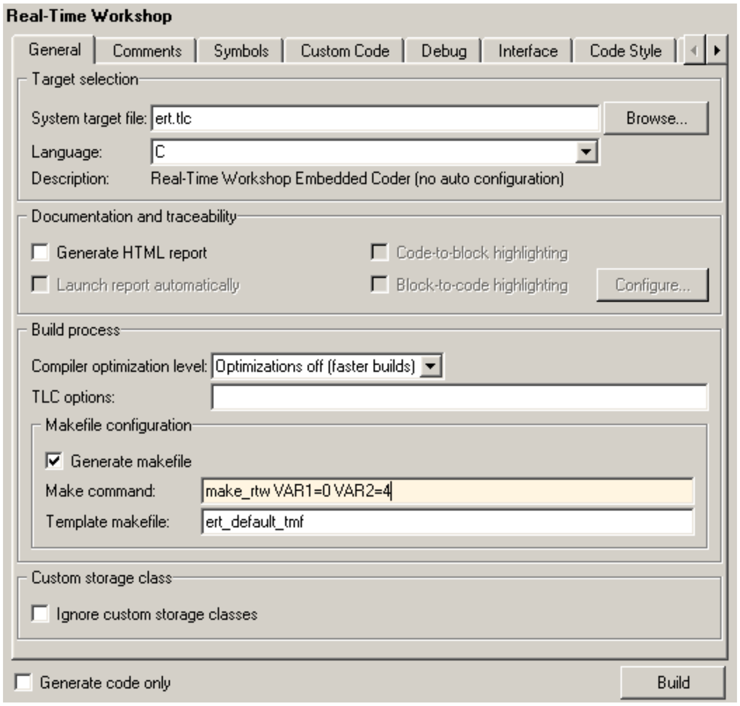
The arguments are defined as:

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



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

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


Real-Time Workshop

Target selection

System target file: ert.tlc

Language: C

Description: Real-Time Workshop Embedded Coder (no auto configuration)

Documentation and traceability

☐ Generate HTML report

☐ Launch report automatically

☐ Code-to-block highlighting

☐ Block-to-code highlighting

Build process

Compiler optimization level: Optimizations off (faster builds)

TLC options:

Makefile configuration

☑ Generate makefile

Make command: make_rtw VAR1=0 VAR2=4

Template makefile: ert_default_tmf

Custom storage class

☐ Ignore custom storage classes

☐ Generate code only

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

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

Applications for STF_make_rtw_hook.m

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

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

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

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

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

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

Using STF_make_rtw_hook.m for Your Build Procedure

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

1. Copy ert_make_rtw_hook.m to a directory in the MATLAB path, and rename it in accordance with the naming conventions described in "File and Function Naming Conventions" on page 5-9. For example, to use it with the GRT target grt.tlc, rename it to grt_make_rtw_hook.m.

2. Rename the ert_make_rtw_hook function within the file to match the filename.

3. Implement the hooks that you require by adding code to the appropriate case statements within the switch hookMethod statement. See "Auto-Configuring Models for Code Generation" on page 5-22 for an example.

Customizing the Target Build Process with sl_customization.m

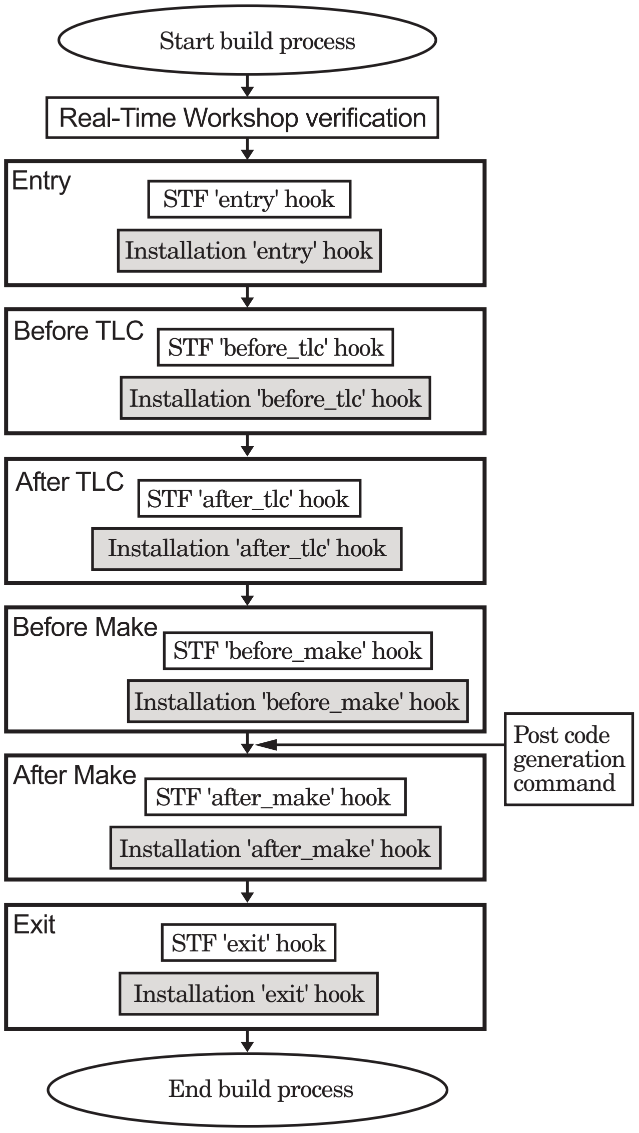
In this section...

Overview

The Simulink customization file `sl_customization.m` is a mechanism that allows you to use M-code to perform customizations of the standard Simulink user interface. Simulink reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see "Customizing the Simulink User Interface" in the Simulink documentation.

The `sl_customization.m` file can be used to register installation-specific hook functions to be invoked during the Real-Time Workshop build process. The hook functions that you register through `sl_customization.m` complement System Target File (STF) hooks (described in "Customizing the Target Build Process with the STF_make_rtw Hook File" on page 5-9) and post-code generation commands (described in "Customizing Post Code Generation Build Processing" in the Real-Time Workshop documentation).

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


Start build process
Real-Time Workshop verification
Entry
STF 'entry' hook
Installation 'entry' hook
Before TLC
STF 'before_tlc' hook
Installation 'before_tlc' hook
After TLC
STF 'after_tlc' hook
Installation 'after_tlc' hook
Before Make
STF 'before_make' hook
Installation 'before_make' hook
Post code generation command
After Make
STF 'after_make' hook
Installation 'after_make' hook
Exit
STF 'exit' hook
Installation 'exit' hook
End build process

Registering Build Process Hook Functions Using sl_customization.m

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

```
function sl_customization(cm)
```

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

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

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

- `addUserHook(hObj, hookType, hook)`

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

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

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

```
sl_refresh_customizations
```

Variables Available for sl_customization.m Hook Functions

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

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

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

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

Example Build Process Customization Using sl_customization.m

The `sl_customization.m` file shown in Example 1: sl_customization.m for Real-Time Workshop Build Process Customizations on page 5-20 uses the `addUserHook` method to specify installation-specific build process hooks to be invoked at the `'entry'` and `'after_tlc'` stages of the Real-Time Workshop build. For the hook function source code, see Example 2: CustomRTWEntryHook.m on page 5-21 and Example 3: CustomRTWPostProcessHook.m on page 5-21.

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

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

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

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

end
```

Example 2: CustomRTWEntryHook.m

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

Example 3: CustomRTWPostProcessHook.m

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

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

```
>> rtwdemo_udt

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

Auto-Configuring Models for Code Generation

In this section...

Overview

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

Utilities for Accessing Model Configuration Properties

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

Using set_param to Set Model Parameters

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

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

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

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

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

Automatic Model Configuration Using ert_make_rtw_hook

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

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

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

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

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

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

        option = LocalParseArgList(buildArgs);

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

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

- The first argument is either a string specifying a model name, for `'entry'` hook invocation, or a configuration set handle, for target selection invocation.
- The second argument is a string specifying a configuration mode, which is extracted from the `buildArgs` argument to `ert_make_rtw_hook.m` (see "STF_make_rtw_hook.m Function Prototype and Arguments" on page 5-11). In the example implementation, the configuration mode is either `'optimized_floating_point'` or `'optimized_fixed_point'`. The

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

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

ert_make_rtw_hook Limitation

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

Using the Auto-Configuration Utilities

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

1. Set up the example `ert_make_rtw_hook.m` as your *`STF`*`_make_rtw_hook` file (see "Customizing the Target Build Process with the STF_make_rtw Hook File" on page 5-9).

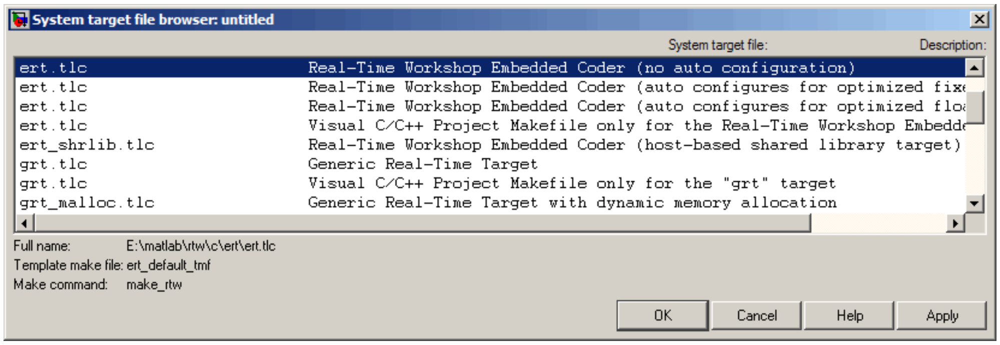
2. Reconfigure the `set_param` calls within `ert_config_opt.m` to suit your application needs.

Generating Efficient Code with Optimized ERT Targets

In this section...

Overview

To make it easier for you to generate code that is optimized for your target hardware, Real-Time Workshop Embedded Coder provides three variants of the ERT target. These targets are based on a common system target file, `ert.tlc`. They are displayed in the System Target File Browser as shown in the figure below.



```
ert.tlc            Real-Time Workshop Embedded Coder (no auto configuration)
ert.tlc            Real-Time Workshop Embedded Coder (auto configures for optimized fixe
ert.tlc            Real-Time Workshop Embedded Coder (auto configures for optimized flo
ert.tlc            Visual C/C++ Project Makefile only for the Real-Time Workshop Embedde
ert_shrlib.tlc     Real-Time Workshop Embedded Coder (host-based shared library target)
grt.tlc            Generic Real-Time Target
grt.tlc            Visual C/C++ Project Makefile only for the "grt" target
grt_malloc.tlc     Generic Real-Time Target with dynamic memory allocation
```

Full name: E:\matlab\rtw\c\ert\ert.tlc
Template make file: ert_default_tmf
Make command: make_rtw

The ERT target variants differ with respect to:

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

Pane" in the Simulink documentation and "Configuring Hardware Properties and Constraints" in the Real-Time Workshop documentation for full details on the **Hardware Implementation** pane).

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

Default ERT Target

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

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

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

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

Optimized Fixed-Point ERT Target

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

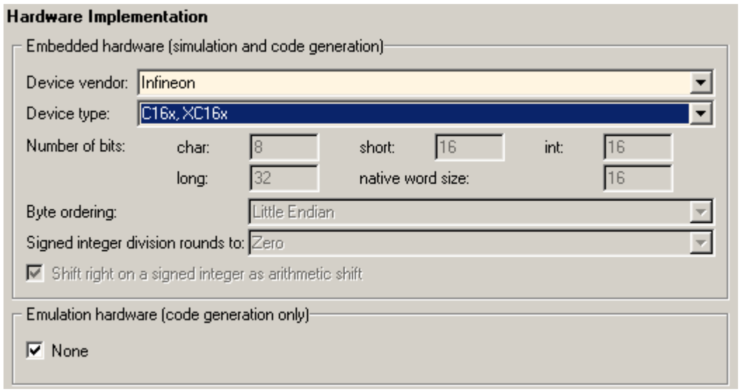
```
Real-Time Workshop Embedded Coder (auto configures for optimized fixed-point code)
```

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

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

You can, if desired, customize the option settings in the auto-configuration file, file, `ert_config_opt.m`. See "Auto-Configuring Models for Code Generation" on page 5-22 for a complete description of the auto-configuration mechanism.

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



Hardware Implementation

Embedded hardware (simulation and code generation)

Device vendor: Infineon

Device type: C16x, XC16x

Number of bits: char: 8 short: 16 int: 16

long: 32 native word size: 16

Byte ordering: Little Endian

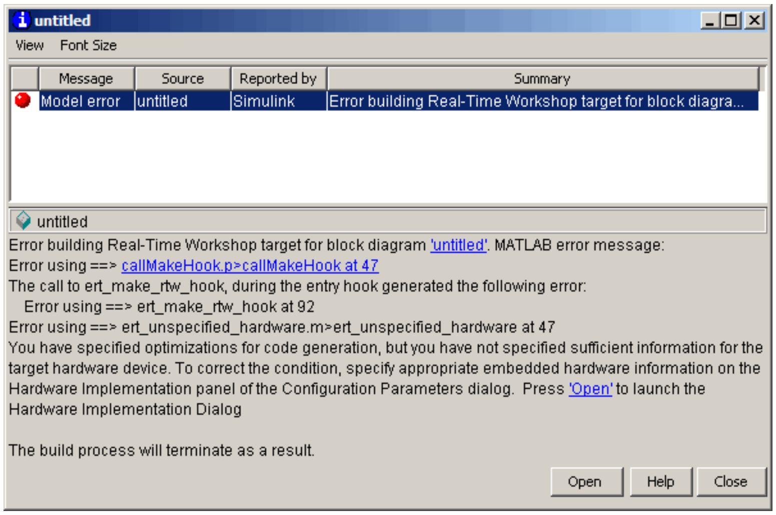
Signed integer division rounds to: Zero

☑ Shift right on a signed integer as arithmetic shift

Emulation hardware (code generation only)

☑ None

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



Message	Source	Reported by	Summary
Model error	untitled	Simulink	Error building Real-Time Workshop target for block diagra...

```
untitled
Error building Real-Time Workshop target for block diagram 'untitled'. MATLAB error message:
Error using ==> callMakeHook.p>callMakeHook at 47
The call to ert_make_rtw_hook, during the entry hook generated the following error:
   Error using ==> ert_make_rtw_hook at 92
Error using ==> ert_unspecified_hardware.m>ert_unspecified_hardware at 47
You have specified optimizations for code generation, but you have not specified sufficient information for the
target hardware device. To correct the condition, specify appropriate embedded hardware information on the
Hardware Implementation panel of the Configuration Parameters dialog.  Press 'Open' to launch the
Hardware Implementation Dialog

The build process will terminate as a result.
```

Optimized Floating-Point ERT Target

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

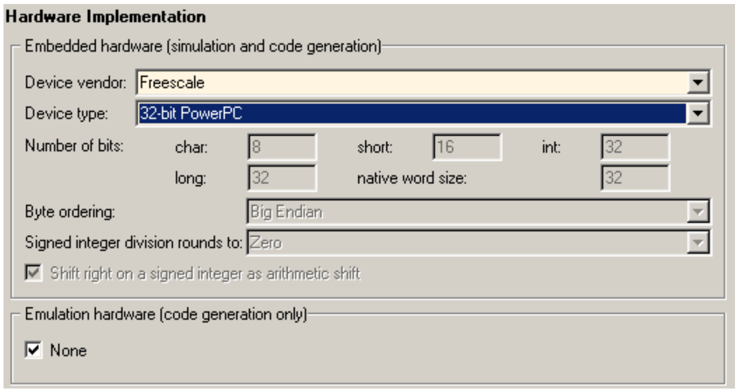
```
Real-Time Workshop Embedded Coder (auto configures for optimized floating-point code)
```

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

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

You can, if desired, customize the option settings in the auto-configuration file, file, `ert_config_opt.m`. See "Auto-Configuring Models for Code Generation" on page 5-22 for a complete description of the auto-configuration mechanism.

The optimized floating-point ERT target requires specification of target hardware characteristics prior to code generation. Before generating code, you should select the desired **Device vendor** and **Device type** (or define a `Custom` device type with **Device vendor** set to `Generic`) in the upper panel of the **Hardware Implementation** pane of the Configuration Parameters dialog box, and set the other properties appropriately for your target. In the figure below, the **Device vendor** and **Device type** fields are configured for the Freescale 32-bit PowerPC family of microprocessors.


Hardware Implementation

Embedded hardware (simulation and code generation)

Device vendor: Freescale

Device type: 32-bit PowerPC

Number of bits: char: 8 short: 16 int: 32

long: 32 native word size: 32

Byte ordering: Big Endian

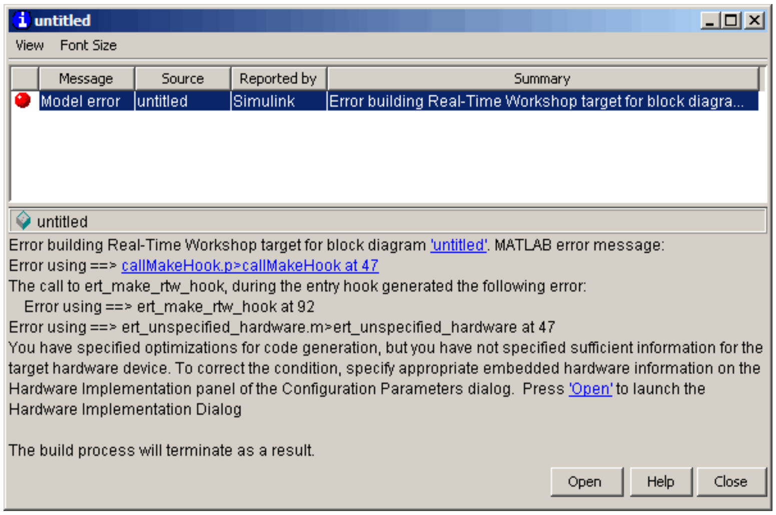
Signed integer division rounds to: Zero

☑ Shift right on a signed integer as arithmetic shift

Emulation hardware (code generation only)

☑ None

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



Model error | untitled | Simulink | Error building Real-Time Workshop target for block diagra...

untitled

```
Error building Real-Time Workshop target for block diagram 'untitled'. MATLAB error message:
Error using ==> callMakeHook.p>callMakeHook at 47
The call to ert_make_rtw_hook, during the entry hook generated the following error:
   Error using ==> ert_make_rtw_hook at 92
Error using ==> ert_unspecified_hardware.m>ert_unspecified_hardware at 47
You have specified optimizations for code generation, but you have not specified sufficient information for the
target hardware device. To correct the condition, specify appropriate embedded hardware information on the
Hardware Implementation panel of the Configuration Parameters dialog.  Press 'Open' to launch the
Hardware Implementation Dialog

The build process will terminate as a result.
```

Using the Optimized ERT Targets

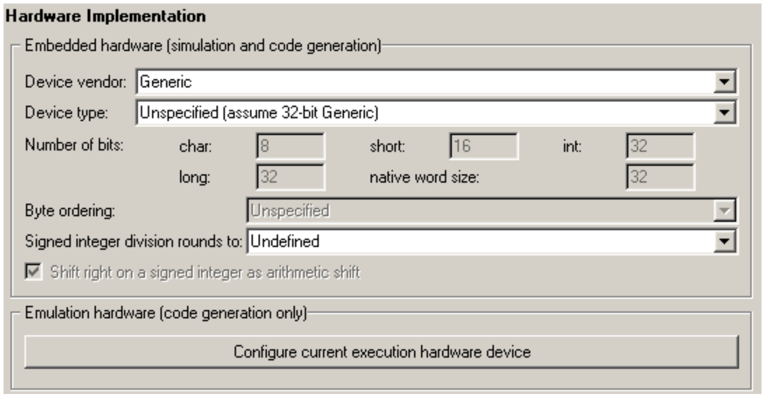
This section describes how to use the optimized ERT targets in code generation.

Configuring Hardware Implementation Properties

Before using one of the optimized versions of the ERT targets, make sure that you have specified the **Hardware Implementation** properties for the model's active configuration set correctly. If this is not done properly, an error message displays at the start of the code generation process and the build terminates.

To avoid such problems, select the desired **Device vendor** and **Device type** (or define a `Custom` device type with **Device vendor** set to `Generic`) in the upper panel of the **Hardware Implementation** pane of the Configuration Parameters dialog box, and set the other properties appropriately for your target (see "Optimized Fixed-Point ERT Target" on page 5-27 and "Optimized Floating-Point ERT Target" on page 5-29). Do not leave the **Device type** unspecified.

Note that if your model was created prior to MATLAB Release 14 and has not yet been updated, the **Device vendor** and **Device type** default to `Generic` and `Unspecified (assume 32-bit Generic)`, and the **Emulation hardware** properties (in the lower section of the **Hardware Implementation** pane) are in an undefined state. This condition is indicated by the presence of a button labeled **Configure current execution hardware device**, as shown in this figure.



Hardware Implementation

Embedded hardware (simulation and code generation)

Device vendor: Generic

Device type: Unspecified (assume 32-bit Generic)

Number of bits: char: 8 short: 16 int: 32

long: 32 native word size: 32

Byte ordering: Unspecified

Signed integer division rounds to: Undefined

☑ Shift right on a signed integer as arithmetic shift

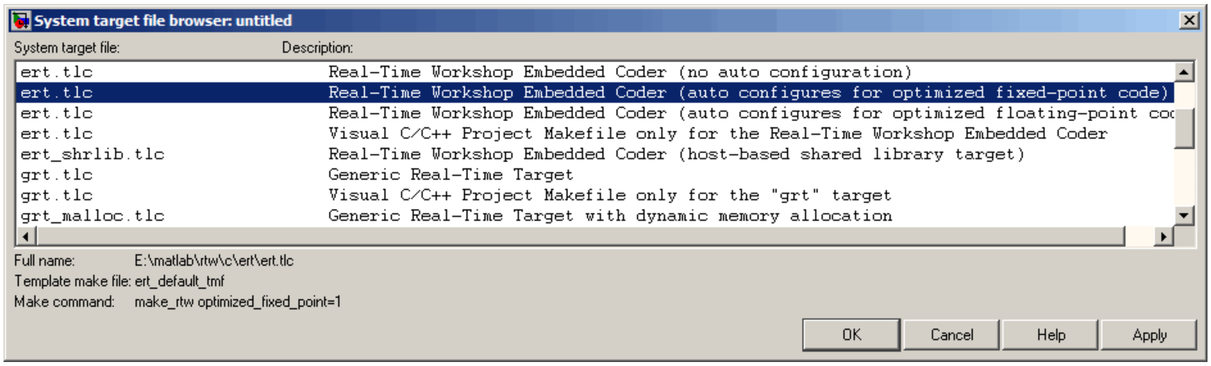
Emulation hardware (code generation only)

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

Generating Code

To generate code using one of the optimized ERT targets:

1 From the **Real-Time Workshop** pane of the Configuration Parameters dialog box, open the System Target File Browser and select the desired target. This figure shows the browser with the optimized fixed-point ERT target selected.



System target file browser: untitled

System target file:	Description:
ert.tlc	Real-Time Workshop Embedded Coder (no auto configuration)
ert.tlc	Real-Time Workshop Embedded Coder (auto configures for optimized fixed-point code)
ert.tlc	Real-Time Workshop Embedded Coder (auto configures for optimized floating-point coc
ert.tlc	Visual C/C++ Project Makefile only for the Real-Time Workshop Embedded Coder
ert_shrlib.tlc	Real-Time Workshop Embedded Coder (host-based shared library target)
grt.tlc	Generic Real-Time Target
grt.tlc	Visual C/C++ Project Makefile only for the "grt" target
grt_malloc.tlc	Generic Real-Time Target with dynamic memory allocation

Full name: E:\matlab\rtw\c\ert\ert.tlc
Template make file: ert_default_tmf
Make command: make_rtw optimized_fixed_point=1

2 When you click **Apply** or **OK** to apply the target selection, the auto-configuration code executes. This is reported in a message similar to the following:

```
*** Auto configuring 'optimized_fixed_point' for model 'untitled' as specified by:
ert_config_opt.m
*** Overwriting model settings if they do not yield optimized code.
```

3 Initiate the build process.

4 If your model's **Hardware Implementation** parameters are not configured correctly, an error message is displayed. If the error appears, see "Configuring Hardware Implementation Properties" on page 5-31 to learn how to correct the problem, and then retry Step 3.

5 During code generation, the auto-configuration code executes a second time, and the auto-configuration message displayed in Step 2 appears again.

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

Custom File Processing

In this section...

Overview

This section describes Real-Time Workshop Embedded Coder *custom file processing* (CFP) features. Custom file processing simplifies generation of custom source code by letting you

- Generate virtually any type of source (`.c` or `.cpp`) or header (`.h`) file. Using a *custom file processing template* (CFP template), you can control how code is emitted to the standard generated model files (for example, *`model`*`.c` or `.cpp`, *`model`*`.h`) or generate files that are independent of model code.
- Organize generated code into sections (such as includes, typedefs, functions, and more). Your CFP template can emit code (for example, functions), directives (such as `#define` or `#include` statements), or comments into each section as required.
- Generate custom *file banners* (comment sections) at the start and end of generated code files.
- Generate code to call model functions such as *`model`*`_initialize`, *`model`*`_step`, and so on.
- Generate code to read and write model inputs and outputs.

- Generate a main program module.
- Obtain information about the model and the files being generated from it.

Custom File Processing Components

The custom file processing features discussed in this section are based on the following interrelated components:

- *Code generation template* (CGT) files: A CGT file defines the top-level organization and formatting of generated code. CGT files are described in "Code Generation Template (CGT) Files" on page 5-37.
- The *code template API*: a high-level Target Language Compiler (TLC) API that provides functions that let you organize code into named sections and subsections of generated source and header files. The code template API also provides utilities that return information about generated files, generate standard model calls and perform other useful functions. See "Code Template API Summary" on page 5-52.
- *Custom file processing (CFP) templates*: A CFP template is a TLC file that manages the process of custom code generation. The primary purpose of a CFP template is to assemble code to be generated into buffers, and to call the code template API to emit the buffered code into specified sections of generated source and header files. A CFP template interacts with a CGT file, which defines the ordering of major sections into which code is emitted. CFP templates and their applications are described in "Using Custom File Processing (CFP) Templates" on page 5-41.

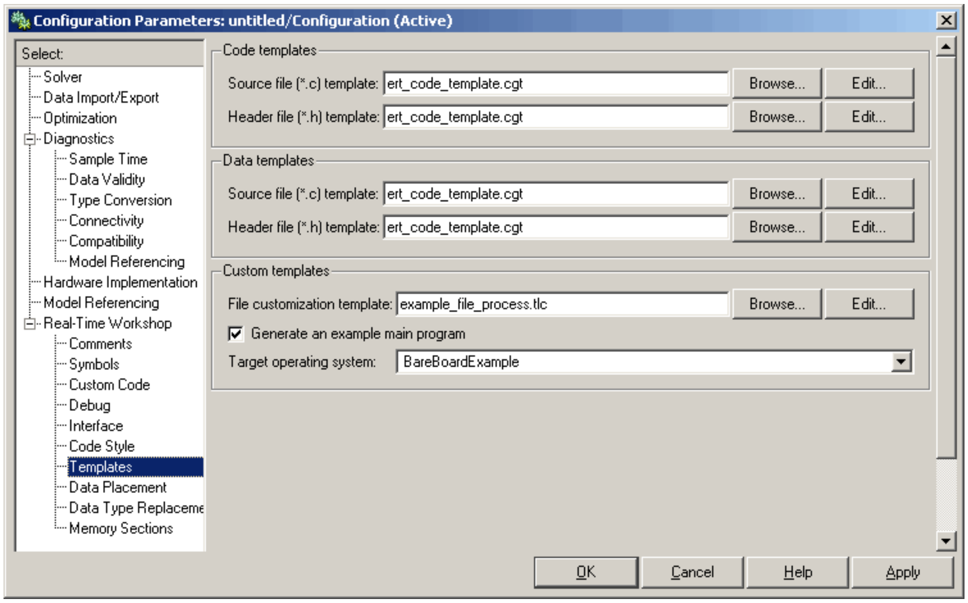
Understanding of TLC programming is required to use CFP templates. See the Target Language Compiler document to learn the basics.

Custom File Processing User Interface Options

Use of custom file processing features requires creation of CGT files and/or CFP templates. Usually, these files are based on default templates provided by Real-Time Workshop Embedded Coder. Once you have created your templates, you must integrate them into the code generation process.

The **Templates** pane of the **Real-Time Workshop** properties of a model configuration set lets you select and edit CGT files and CFP templates, and specify their use in the code generation process. Real-Time

Workshop/Templates Pane on page 5-36 shows this pane, with all options configured for their defaults.


Configuration Parameters: untitled/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
  - Sample Time
  - Data Validity
  - Type Conversion
  - Connectivity
  - Compatibility
  - Model Referencing
- Hardware Implementation
- Model Referencing
- Real-Time Workshop
  - Comments
  - Symbols
  - Custom Code
  - Debug
  - Interface
  - Code Style
  - Templates
  - Data Placement
  - Data Type Replaceme
  - Memory Sections

Code templates
Source file (*.c) template: ert_code_template.cgt
Header file (*.h) template: ert_code_template.cgt

Data templates
Source file (*.c) template: ert_code_template.cgt
Header file (*.h) template: ert_code_template.cgt

Custom templates
File customization template: example_file_process.tlc
[✓] Generate an example main program
Target operating system: BareBoardExample

Real-Time Workshop/Templates Pane

The options related to custom file processing are:

- The **Source file (.c) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating source (`.c` or `.cpp`) files. This file must be located on the MATLAB path.
- The **Header file (.h) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating header (`.h`) files. This file must be located on the MATLAB path.

  By default, the template for both source and header files is *`matlabroot`*`/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
- The **File customization template** edit field in the **Custom templates** section. This field specifies the name of a CFP

template file to use when generating code files. This file must be located on the MATLAB path. The default CFP template is `matlabroot/toolbox/rtw/targets/ecoder/example_file_process.tlc`.

Each of these fields has associated **Browse** and **Edit** buttons. **Browse** lets you navigate to and select an existing CFP template or CGT file. **Edit** opens the specified CFP template into the MATLAB editor, where you can customize it.

Code Generation Template (CGT) Files

CGT files have a number of applications:

- The simplest application is generation of custom file banners (comments sections) in code files. To do this, no knowledge of the details of the CGT file structure is required; see "Generating Custom File Banners" on page 5-55.
- Some of the advanced features described in the Module Packaging Features document utilize CGT files. Refer to that document for information.
- When generating custom code using a CFP template, a CGT file is required. Correct use of CFP templates requires understanding of the CGT file structure, although in many cases it is possible to use the default CGT file without modification.

Default CGT file

Real-Time Workshop Embedded Coder provides a default CGT file: `matlabroot\toolbox\rtw\targets\ecoder\ert_code_template.cgt`.

You should base your custom CGT files on the default file.

CGT File Structure

A CGT file consists of three sections:

Header Section. This section is optional. It contains comments and tokens for use in generating a custom header banner. "Generating Custom File Banners" on page 5-55 gives details on custom banner generation.

Code Insertion Section. This section is required. It contains tokens that define an ordered partitioning of the generated code into a number of sections (such as `Includes` and `Defines` sections). Tokens have the form

```
%<SectionName>
```

For example,

```
%<Includes>
```

Real-Time Workshop Embedded Coder defines a minimal set of tokens that are required for the generation of C or C++ source or header code. These are *built-in* tokens (see "Built-In Tokens and Sections" on page 5-38). You can also define *custom* tokens and add them to the code insertion section (see "Generating a Custom Section" on page 5-50).

Each token functions as a placeholder for a corresponding section of generated code. The ordering of the tokens defines the order in which the corresponding sections appear on the generated code. The presence of a token in the CGT file does not guarantee that the corresponding section is generated. To generate code into a given section, you must do so explicitly by calling the code template API from a CFP template, as described in "Using Custom File Processing (CFP) Templates" on page 5-41.

The CGT tokens define the high-level organization of generated code. Using the code template API, you can partition each code section into named subsections, as described in "Subsections" on page 5-40.

You can also insert C or C++ comments into the code insertion section, between tokens. Such comments are inserted directly into the generated code.

Trailer Section. This section is optional. It contains comments and tokens for use in generating a custom trailer banner. "Generating Custom File Banners" on page 5-55 gives details on custom banner generation.

Built-In Tokens and Sections

The following code extract shows the code insertion section of the default CGT file, showing the built-in tokens.

```
%% Required tokens.  You can insert comments and other tokens in between them,
%% but do not change their order or remove them.
%%
%<Includes>
%<Defines>
%<Types>
%<Enums>
%<Definitions>
%<Declarations>
%<Functions>
```

Note carefully the following requirements before creating or customizing a CGT file:

- All the built-in tokens are required. None can be removed.
- Built-in tokens must appear in the order shown. The ordering is significant because each successive section can have dependencies on previous sections.
- Only one token can appear per line.
- Tokens must not be repeated.
- Custom tokens can be added to the code insertion section, provided that the previous requirements are not violated.
- Comments can be added to the code insertion section, provided that the previous requirements are not violated.

Built-In CGT Tokens and Corresponding Code Sections on page 5-39 summarizes the built-in tokens and corresponding section names, and describes the code sections.

Built-In CGT Tokens and Corresponding Code Sections

Token / Section Name	Description
`Includes`	`#include` directives section
`Defines`	`#define` directives section
`Types`	`typedef` section. `Typedefs` can depend on any previously defined type

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

Token / Section Name	Description
`Enums`	Enumerated types section
`Definitions`	Place data definitions here (for example, `double x = 3.0;`)
`Declarations`	Data declarations (for example, `extern double x;`)
`Functions`	C or C++ functions

Subsections

It is possible to define one or more named subsections for any section. Some of the built-in sections have predefined subsections. These are summarized in Subsections Defined for Built-In Sections on page 5-40.

It is important to note that the sections and subsections listed in Subsections Defined for Built-In Sections on page 5-40 are emitted, in the order listed, to the source or header file being generated.

The custom section feature lets you define sections in addition to those listed in Subsections Defined for Built-In Sections on page 5-40. See “Generating a Custom Section” on page 5-50 for information on how to do this.

Subsections Defined for Built-In Sections

Section	Subsections	Subsection Description
`Includes`	N/A	
`Defines`	N/A	
`Types`	`IntrinsicTypes`	Intrinsic `typedef` section. Intrinsic types are those that depend only on intrinsic C or C++ types.

Subsections Defined for Built-In Sections (Continued)

Section	Subsections	Subsection Description
`Types`	`PrimitiveTypedefs`	Primitive `typedef` section. Primitive typedefs are those that depend only on intrinsic C or C++ types and on any typedefs previously defined in the `IntrinsicTypes` section.
`Types`	`UserTop`	Any type of code can be placed in this section. You can place code that has dependencies on the previous sections here.
`Types`	`Typedefs`	`typedef` section. Typedefs can depend on any previously defined type
`Enums`	N/A	
`Definitions`	N/A	
`Declarations`	N/A	
`Functions`		C or C++ functions
`Functions`	`CompilerErrors`	`#warning` directives
`Functions`	`CompilerWarnings`	`#error` directives
`Functions`	`Documentation`	Documentation (comment) section
`Functions`	`UserBottom`	Any code can be placed in this section.

Using Custom File Processing (CFP) Templates

The files provided to support custom file processing are

- *`matlabroot`*`\rtw\c\tlc\mw\codetemplatelib.tlc`: A TLC function library that implements the code template API. `codetemplatelib.tlc` also provides the comprehensive documentation of the API in the comments headers preceding each function.
- *`matlabroot`*`\toolbox\rtw\targets\ecoder\example_file_process.tlc`: An example custom file processing (CFP) template, which you should use as the starting point for creating your own CFP templates. Guidelines and examples for creating a CFP template are provided in “Generating

Source and Header Files with a Custom File Processing (CFP) Template" on page 5-43.

- TLC files supporting generation of single-rate and multirate main program modules (see "Customizing Main Program Module Generation" on page 5-48).

Once you have created a CFP template, you must integrate it into the code generation process, using the **File customization template** edit field (see "Custom File Processing User Interface Options" on page 5-35).

Custom File Processing (CFP) Template Structure

A custom file processing (CFP) template imposes a simple structure on the code generation process. The template, in conjunction with a code generation template (CGT) file, partitions the code generated for each file into a number of sections. These sections are summarized in Built-In CGT Tokens and Corresponding Code Sections on page 5-39 and Subsections Defined for Built-In Sections on page 5-40.

Code for each section is assembled in buffers and then emitted, in the order listed, to the file being generated.

To generate a file section, your CFP template must first assemble the code to be generated into a buffer. Then, to emit the section, your template calls the TLC function

```
LibSetSourceFileSection(fileH, section, tmpBuf)
```

where

- `fileH` is a file reference to a file being generated.
- `section` is the code section or subsection to which code is to be emitted. section must be one of the section or subsection names listed in Subsections Defined for Built-In Sections on page 5-40.

  Determine the `section` argument as follows:

  - If Subsections Defined for Built-In Sections on page 5-40 defines no subsections for a given section, use the section name as the `section` argument.

  - If Subsections Defined for Built-In Sections on page 5-40 defines one or more subsections for a given section, you can use either the section name or a subsection name as the `section` argument.
  - If you have defined a custom token denoting a custom section, do not call `LibSetSourceFileSection`. Special API calls are provided for custom sections (see "Generating a Custom Section" on page 5-50).
- `tmpBuf` is the buffer containing the code to be emitted.

There is no requirement to generate all of the available sections. Your template need only generate the sections you require in a particular file.

Note that no legality or syntax checking is performed on the custom code within each section.

The next section, "Generating Source and Header Files with a Custom File Processing (CFP) Template" on page 5-43, provides typical usage examples.

Generating Source and Header Files with a Custom File Processing (CFP) Template

This section walks you through the process of generating a simple source (`.c` or `.cpp`) and header (`.h`) file using the example CFP template. Then, it examines the template and the code generated by the template.

The example CFP template, `example_file_process.tlc`, demonstrates some of the capabilities of the code template API, including

- Generation of simple source (`.c` or `.cpp`) and header (`.h`) files
- Use of buffers to generate file sections for includes, functions, and so on
- Generation of includes, defines, and so on into the standard generated files (for example, *`model`*`.h`)
- Generation of a main program module

Generating Code with a CFP Template

This section sets up a CFP template and configures a model to use the template in code generation. The template generates (in addition to the standard model files) a source file (`timestwo.c` or `.cpp`) and a header file (`timestwo.h`).

You should follow the steps below to become acquainted with the use of CFP templates:

1. Copy the example CFP template, *`matlabroot`*`/toolbox/rtw/targets/ecoder/example_file_process.tlc`, to a directory of your choice. This directory should be located outside the MATLAB directory structure (that is, it should not be under *`matlabroot`*.) Note that this directory must be on the MATLAB path, or on the TLC path. It is good practice to locate the CFP template in the same directory as your system target file, which is guaranteed to be on the TLC path.

2. Rename the copied `example_file_process.tlc` to `test_example_file_process.tlc`.

3. Open `test_example_file_process.tlc` into the MATLAB editor.

4. Uncomment the following line:

   ```
   %%  %assign ERTCustomFileTest = TLC_TRUE
   ```

   It should now read:

   ```
     %assign ERTCustomFileTest = TLC_TRUE
   ```

   If `ERTCustomFileTest` is not assigned as shown, the CFP template is ignored in code generation.

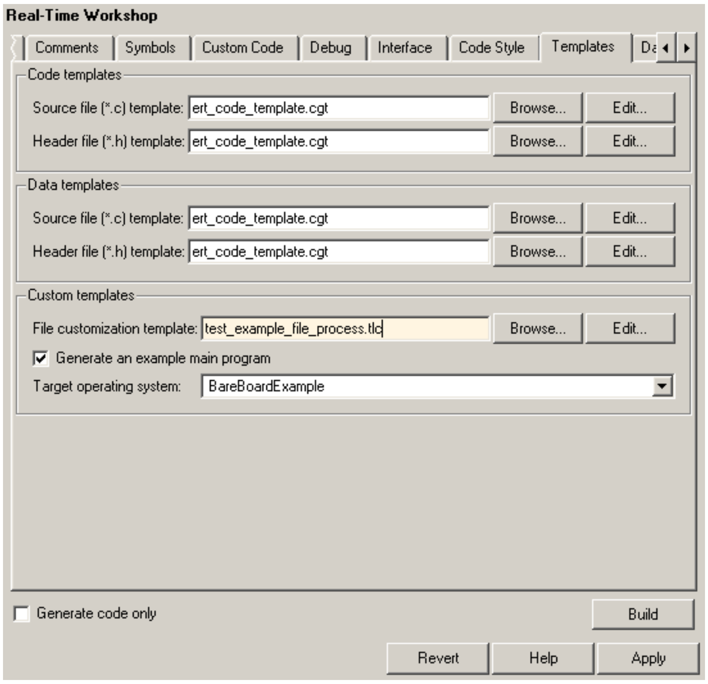
5. Save your changes to the file. Keep `test_example_file_process.tlc` open, so you can refer to it later.

6. Open the `rtwdemo_udt` model.

7. Open the Simulink Model Explorer. Select the active configuration set of the model, and open the **Real-Time Workshop** properties view of the active configuration set.

8. Click on the **Templates** tab.

9 Configure the **File customization template** field as shown below. The `test_example_file_process.tlc` file, which you previously edited, is now specified as the CFP template.



Real-Time Workshop

Code templates
Source file (*.c) template: ert_code_template.cgt
Header file (*.h) template: ert_code_template.cgt

Data templates
Source file (*.c) template: ert_code_template.cgt
Header file (*.h) template: ert_code_template.cgt

Custom templates
File customization template: test_example_file_process.tlc
[✓] Generate an example main program
Target operating system: BareBoardExample

[ ] Generate code only

10 Select the **Generate code only** option.

11 Click **Apply**.

12 Click **Generate code**. During code generation, notice the following message on the MATLAB command window:

```
Warning:  Overriding example ert_main.c!
```

This message is displayed because `test_example_file_process.tlc` generates the main program module, overriding the default action of the ERT target. This is explained in greater detail below.

13 The `rtwdemo_udt` model is configured to generate an HTML code generation report. After code generation completes, view the report. Notice that the **Generated Source Files** list contains the files `timestwo.c`, `timestwo.h`, and `ert_main.c`. These files were generated by the CFP template. The next section examines the template to learn how this was done.

14 Keep the model, the code generation report, and the `test_example_file_process.tlc` file open so you can refer to them in the next section.

Analysis of the Example CFP Template and Generated Code

This section examines excerpts from `test_example_file_process.tlc` and some of the code it generates. You should refer to the comments in `codetemplatelib.tlc` while reading the discussion below.

Generating Code Files. Source (`.c` or `.cpp`) and header (`.h`) files are created by calling `LibCreateSourceFile`, as in the following excerpts:

```
%assign cFile = LibCreateSourceFile("Source", "Custom", "timestwo")
...
%assign hFile = LibCreateSourceFile("Header", "Custom", "timestwo")
```

Subsequent code refers to the files by the file reference returned from `LibCreateSourceFile`.

File Sections and Buffers. The code template API lets you partition the code generated to each file into sections, tagged as `Definitions`, `Includes`, `Functions`, `Banner`, and so on. You can append code to each section as many times as required. This technique gives you a great deal of flexibility in the formatting of your custom code files.

The available file sections, and the order in which they are emitted to the generated file, are summarized in Subsections Defined for Built-In Sections on page 5-40.

For each section of a generated file, use `%openfile` and `%closefile` to store the text for that section in temporary buffers. Then, to write (append) the buffer contents to a file section, call `LibSetSourceFileSection`, passing in the desired section tag and file reference. For example, the following code uses two buffers (`tmwtypesBuf` and `tmpBuf`) to generate two sections (tagged "Includes" and "Functions") of the source file timestwo.c or .cpp (referenced as cFile):

```
 %openfile tmwtypesBuf

 #include "tmwtypes.h"

%closefile tmwtypesBuf

%<LibSetSourceFileSection(cFile,"Includes",tmwtypesBuf)>

 %openfile tmpBuf

 /* Times two function */
 real_T timestwofcn(real_T input) {
   return (input * 2.0);
}

%closefile tmpBuf

%<LibSetSourceFileSection(cFile,"Functions",tmpBuf)>
```

These two sections generate the entire timestwo.c or .cpp file:

```
#include "tmwtypes.h"

/* Times two function */
FLOAT64 timestwofcn(FLOAT64 input)
{
  return (input * 2.0);
}
```

Adding Code to Standard Generated Files. The timestwo.c or .cpp file generated in the previous example was independent of the standard code files generated from a model (for example, *model*.c or .cpp, *model*.h, and so on). You can use similar techniques to generate custom code within the model files. The code template API includes functions to obtain the names of the standard models files and other model-related information. The following excerpt calls LibGetMdlPubHdrBaseName to obtain the correct name for the *model*.h file. It then obtains a file reference and generates a definition in the Defines section of *model*.h:

```
%% Add a #define to the model's public header file model.h

%assign pubName = LibGetMdlPubHdrBaseName()
%assign modelH  = LibCreateSourceFile("Header", "Simulink", pubName)

%openfile tmpBuf

 #define ACCELERATION 9.81

 %closefile tmpBuf

%<LibSetSourceFileSection(modelH,"Defines",tmpBuf)>
```

Examine the generated `rtwdemo_udt.h` file to see the generated `#define` directive.

Customizing Main Program Module Generation. Normally, the ERT target follows the **Generate an example main program** and **Target operating system** options to determine how to generate an `ert_main.c` or `.cpp` module (if any). You can use a CFP template to override the normal behavior and generate a main program module customized for your target environment.

To support generation of main program modules, two TLC files are provided:

- `bareboard_srmain.tlc`: TLC code to generate an example single-rate main program module for a bareboard target environment. Code is generated by a single TLC function, `FcnSingleTaskingMain`.
- `bareboard_mrmain.tlc`: TLC code to generate a multirate main program module for a bareboard target environment. Code is generated by a single TLC function, `FcnMultiTaskingMain`.

In the example CFP template, the following code generates either a single- or multitasking `ert_main.c` or `.cpp` module. The logic depends on information obtained from the code template API calls `LibIsSingleRateModel` and `LibIsSingleTasking`:

```
%% Create a simple main.  Files are located in MATLAB/rtw/c/tlc/mw.

 %if LibIsSingleRateModel() || LibIsSingleTasking()
```

```
    %include "bareboard_srmain.tlc"
    %<FcnSingleTaskingMain()>
  %else
    %include "bareboard_mrmain.tlc"
    %<FcnMultiTaskingMain()>
  %endif
```

Note that `bareboard_srmain.tlc` and `bareboard_mrmain.tlc` use the code template API to generate `ert_main.c` or `.cpp`.

When generating your own main program module, you disable the default generation of `ert_main.c` or `.cpp`. The TLC variable `GenerateSampleERTMain` controls generation of `ert_main.c` or `.cpp`. You can directly force this variable to `TLC_FALSE`. The examples `bareboard_mrmain.tlc` and `bareboard_srmain.tlc` use this technique, as shown in the following excerpt from `bareboard_srmain.tlc`.

```
%if GenerateSampleERTMain
    %assign CompiledModel.GenerateSampleERTMain = TLC_FALSE
    %warning Overriding example ert_main.c!
%endif
```

Alternatively, you can implement a `SelectCallback` function for your target. A `SelectCallback` function is an M function that is triggered during model loading, and also when the user selects a target with the System Target File browser. Your `SelectCallback` function should deselect and disable the **Generate an example main program** option. This prevents the TLC variable `GenerateSampleERTMain` from being set to TLC_TRUE.

See the "rtwgensettings Structure" section of the Developing Embedded Targets document for information on creating a `SelectCallback` function.

The following code illustrates how to deselect and disable the **Generate an example main program** option in the context of a `SelectCallback` function.

```
slConfigUISetVal(hDlg, hSrc, 'GenerateSampleERTMain', 'off');
slConfigUISetEnabled(hDlg, hSrc, 'GenerateSampleERTMain',0);
```

Note Creation of a main program for your target environment requires some customization; for example, in a bareboard environment you need to attach `rt_OneStep` to a timer interrupt. It is expected that you will customize either the generated code, the generating TLC code, or both. See "Guidelines for Modifying the Main Program" on page 1-14 and "Guidelines for Modifying rt_OneStep" on page 1-20 for further information.

Generating a Custom Section

You can define custom tokens in a CGT file and direct generated code into an associated built-in section. This feature gives you additional control over the formatting of code within each built-in section. For example, you could add subsections to built-in sections that do not already define any subsections. All custom sections must be associated with one of the built-in sections: `Includes`, `Defines`, `Types`, `Enums`, `Definitions`, `Declarations`, or `Functions`. To create custom sections, you must

- Add a custom token to the code insertion section of your CGT file.
- In your CFP file:
  - Assemble code to be generated to the custom section into a buffer.
  - Declare an association between the custom section and a built-in section, with the code template API function `LibAddSourceFileCustomSection`.
  - Emit code to the custom section with the code template API function `LibSetSourceFileCustomSection`.

The following code examples illustrate the addition of a custom token, `Myincludes`, to a CGT file, and the subsequent association of the custom section `Myincludes` with the built-in section `Includes` in a CFP file.

First, add the token `Myincludes` to the code insertion section of your CGT file. For example:

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

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

Next, in the CFP file, generate include directives into a buffer. For example:

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

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

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

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

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

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

```
#include "rtwdemo_udt.h"
#include "rtwdemo_udt_private.h"
#include "moretables1.h"
#include "moretables2.h"
```

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

Code Template API Summary

Code Template API Functions on page 5-52 summarizes the code template API. See the source code in `codetemplatelib.tlc` for detailed information on the arguments, return values, and operation of these calls.

Code Template API Functions

Function	Description
`LibGetNumSourceFiles`	Returns the number of created source files (`.c` or `.cpp` and `.h`).
`LibGetSourceFileTag`	Returns `<filename>_h` and `<filename>_c` for header and source files, respectively, where `filename` is the name of the model file.
`LibCreateSourceFile`	Creates a new C or C++ file and returns its reference. If the file already exists, simply returns its reference.
`LibGetSourceFileFromIdx`	Returns a model file reference based on its index. This is useful for a common operation on all files, such as to set the leading file banner of all files.
`LibSetSourceFileSection`	Adds to the contents of a specified section within a specified file (see also "Custom File Processing (CFP) Template Structure" on page 5-42).
`LibIndentSourceFile`	Indents a file with the `c_indent` utility of Real-Time Workshop (from within the TLC environment).
`LibCallModelInitialize`	Returns code for calling the model's *`model`*`_initialize` function (valid for ERT only).
`LibCallModelStep`	Returns code for calling the model's *`model`*`_step` function (valid for ERT only).

Code Template API Functions (Continued)

Function	Description
`LibCallModelTerminate`	Returns code for calling the model's *`model`*`_terminate` function (valid for ERT only).
`LibCallSetEventForThisBaseStep`	Returns code for calling the model's set events function (valid for ERT only).
`LibWriteModelData`	Returns data for the model (valid for ERT only).
`LibSetRTModelErrorStatus`	Returns the code to set the model error status.
`LibGetRTModelErrorStatus`	Returns the code to get the model error status.
`LibIsSingleRateModel`	Returns true if model is single rate and false otherwise.
`LibGetModelName`	Returns name of the model (no extension).
`LibGetMdlSrcBaseName`	Returns the name of model's main source file (for example, *`model`*`.c` or `.cpp`).
`LibGetMdlPubHdrBaseName`	Returns the name of model's public header file (for example, *`model`*`.h`).
`LibGetMdlPrvHdrBaseName`	Returns the name of the model's private header file (for example, *`model`*`_private.h`).
`LibIsSingleTasking`	Returns `true` if the model is configured for singletasking execution.
`LibWriteModelInput`	Returns the code to write to a particular root input (that is, a model inport block). (valid for ERT only).
`LibWriteModelOutput`	Returns the code to write to a particular root output (that is, a model outport block). (valid for ERT only).

Code Template API Functions (Continued)

Function	Description
`LibWriteModelInputs`	Returns the code to write to root inputs (that is, all model inport blocks). (valid for ERT only)
`LibWriteModelOutputs`	Returns the code to write to root outputs (that is, all model outport blocks). (valid for ERT only).
`LibNumDiscreteSampleTimes`	Returns the number of discrete sample times in the model.
`LibSetSourceFileCodeTemplate`	Set the code template to be used for generating a specified source file.
`LibSetSourceFileOutputDirectory`	Set the directory into which a specified source file is to be generated.
`LibAddSourceFileCustomSection`	Add a custom section to a source file. The custom section must be associated with one of the built-in (required) sections: `Includes`, `Defines`, `Types`, `Enums`, `Definitions`, `Declarations`, or `Functions`.
`LibSetSourceFileCustomSection`	Adds to the contents of a specified custom section within a specified file. The custom section must have been previously created with `LibAddSourceFileCustomSection`.
`LibGetSourceFileCustomSection`	Returns the contents of a specified custom section within a specified file.
`LibSetCodeTemplateComplianceLevel`	This function must be called from your CFP template before any other code template API functions are called. Pass in `2` as the `level` argument.

Generating Custom File Banners

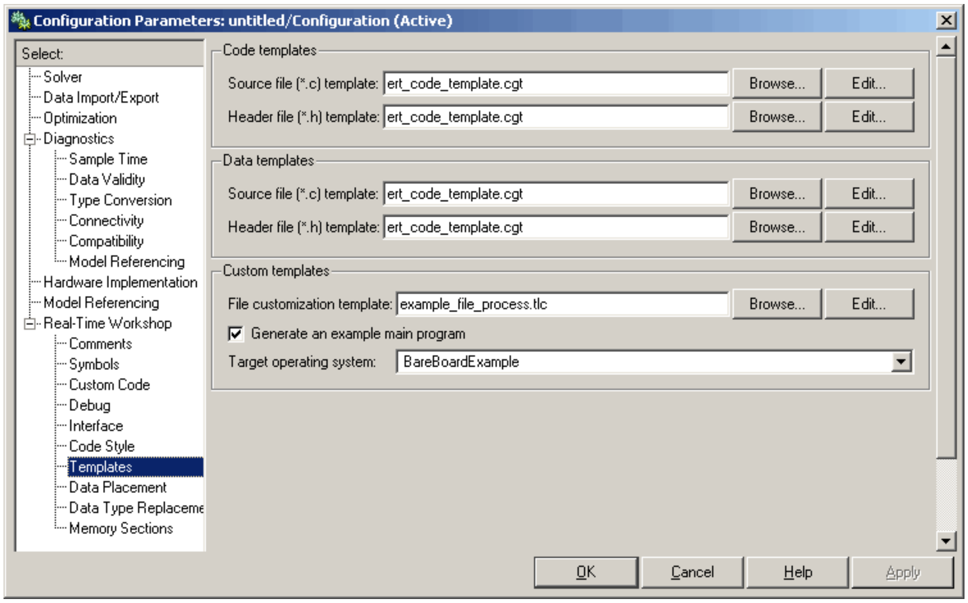
Using code generation template (CGT) files, you can specify custom *file banners* to be inserted into generated code files. File banners are comment sections in the header and trailer portions of a generated file. You can use these banners to add a company copyright statement, specify a special version symbol for your configuration management system, remove time stamps, and for many other purposes. These banners can contain non US-ASCII characters, which are propagated to the generated code.

The recommended technique for specifying file banners is to create a custom CGT file with a customized banner section. During the build process, an executable TLC file is created from the CGT file. This TLC file is then invoked during the code generation process.

You do not need to be familiar with TLC programming to generate custom banners. Generally, you simply need to modify example files supplied with the ERT target.

Note Prior releases supported direct use of customized TLC file as banner templates. These were specified with the **Source file (.c) banner template** and **Header file (.h) banner template** options of the ERT target. Direct use of a TLC file for this purpose is still supported for backward compatibility, but you should now use CGT files for this purpose instead.

File banner generation is supported by the options in the **Code templates** section of the **Templates** pane of the **Real-Time Workshop** properties of a configuration set (shown in ERT Templates Options on page 5-56).



Code templates

Source file (*.c) template: ert_code_template.cgt

Header file (*.h) template: ert_code_template.cgt

Data templates

Source file (*.c) template: ert_code_template.cgt

Header file (*.h) template: ert_code_template.cgt

Custom templates

File customization template: example_file_process.tlc

[✓] Generate an example main program

Target operating system: BareBoardExample

ERT Templates Options

The options related to file banner generation are

- **Source file (.c) template**: CGT file to use when generating source (`.c` or `.cpp`) files. This file must be located on the MATLAB path.
- **Header file (.h) template**: CGT file to use when generating header (`.h`) files. This file must be located on the MATLAB path. This can be the same template specified in the **Source file (.c) template** field, in which case identical banners are generated in source and header files.

  By default, the template for both source and header files is `*matlabroot*/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
- Each of these fields has associated **Browse** and **Edit** buttons. **Browse** lets you navigate to and select an existing CGT file for use as a template. **Edit** opens the specified file into the MATLAB editor, where you can customize it.

Creating a Custom File Banner Template

The recommended procedure for customizing a CGT for custom file banner generation is to make a local copy of the default code template and edit it, as follows:

1. Activate the configuration set you want to work with.
2. Open the **Real-Time Workshop** properties view of the active configuration set.
3. Click on the **Templates** tab (see ERT Templates Options on page 5-56).
4. By default, the code template specified in the **Source file (.c) template** and **Header file (.h) template** fields is `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
5. If you want to use a different template as your starting point, use the **Browse** button to locate and select a CGT file.
6. Click the **Edit** button to open the CGT file into the MATLAB editor.
7. Save a local copy of the CGT file. Store the copy in a directory that is not inside the MATLAB directory structure. Note that this directory must be on the MATLAB path. If necessary, add the directory to the MATLAB path.
8. If you intend to use the CGT file in conjunction with a custom target, it is good practice to locate the CGT file in a folder under your target's root directory.
9. It is also good practice to rename your local copy of the CGT file. When you rename the CGT file, make sure to edit the associated **Source file (.c) template** or **Header file (.h) template** field to match the new filename.
10. Edit and customize the CGT file as needed (See "Customizing a Code Generation Template (CGT) File for Custom Banner Generation" on page 5-58). Before exiting the MATLAB editor, save your changes to the CGT file.
11. Click **Apply** to update the configuration set.
12. Save your model.

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

Customizing a Code Generation Template (CGT) File for Custom Banner Generation

This section describes the sections of a CGT file you need to modify for custom file banner generation. For a more detailed description of CGT files, see "Code Generation Template (CGT) Files" on page 5-37.

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

- Header section: This section contains comments and tokens for use in generating a header banner. The header banner precedes any C or C++ code generated by the model. If the header section is omitted, no header banner is generated. The following is the default header section provided with the default CGT file, `*matlabroot*\toolbox\rtw\targets\ecoder\ert_code_template.cgt`.

```
%% Custom file banner (optional)
%%
/*
 * File: %<FileName>
 *
 * Real-Time Workshop code generated for Simulink model %<ModelName>.
 *
 * Model version                        : %<ModelVersion>
 * Real-Time Workshop file version      : %<RTWFileVersion>
 * Real-Time Workshop file generated on : %<RTWFileGeneratedOn>
 * TLC version                          : %<TLCVersion>
 * C source code generated on           : %<SourceGeneratedOn>
 */
```

- Trailer section: This section contains comments and tokens for use in generating a trailer banner. The trailer banner follows any C or C++ code generated by the model. If the trailer section is omitted, no trailer banner is generated. The following is the default trailer section provided in the default CGT file.

  ```
  %% Custom file trailer (optional)
  %%
  /* File trailer for Real-Time Workshop generated code.
  *
  * [EOF]
  */
  ```

The header and trailer sections typically use TLC variables (such as `%<ModelVersion>`) as tokens. During code generation, tokens are replaced with values in the generated code. See Summary of Tokens for File Banner Generation on page 5-60 for a list of available tokens.

The following code excerpt shows a modified banner section based on the default CGT. This template inserts a copyright notice into the banner.

```
%% Custom file banner (optional)
%%
/*
 * File: %<FileName>
 * ---------------------------------------------------
 * Copyright 2003 ABC Corporation, Inc.
 * ---------------------------------------------------
 * Real-Time Workshop code generated for Simulink model %<ModelName>.
 *
 * Model version                        : %<ModelVersion>
 * Real-Time Workshop file version      : %<RTWFileVersion>
 * Real-Time Workshop file generated on : %<RTWFileGeneratedOn>
 * TLC version                          : %<TLCVersion>
 * C source code generated on           : %<SourceGeneratedOn>
 *
 */
```

The following code excerpt shows a file banner generated from the `rtwdemo_udt` model using the above template.

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

Summary of Tokens for File Banner Generation

`FileName`	Name of the generated file (for example, `"rtwdemo_udt.c"`).
`FileType`	Either `"source"` or `"header"`. Designates whether generated file is a `.c` or `.cpp` file or an `.h` file.
`FileTag`	Given filenames `file.c` or `.cpp` and `file.h`, the file tags are `"file_c"` and `"file_h"`, respectively.
`ModelName`	Name of generating model.
`ModelVersion`	Version number of model.
`RTWFileVersion`	Version number of *`model`*`.rtw` file.
`RTWFileGeneratedOn`	Timestamp of *`model`*`.rtw` file.
`TLCVersion`	Version of Target Language Compiler.
`SourceGeneratedOn`	Timestamp of generated file.

Optimizing Your Model with Configuration Wizard Blocks and Scripts

In this section...

Overview

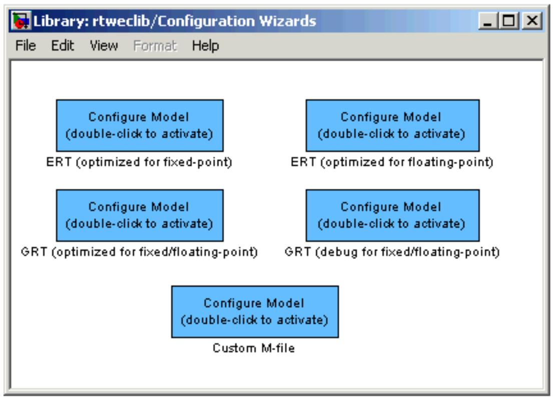
Real-Time Workshop Embedded Coder provides a library of *Configuration Wizard* blocks and scripts to help you configure and optimize code generation from your models quickly and easily.

The library provides a Configuration Wizard block you can customize, and four preset Configuration Wizard blocks.

Block	Description
Custom M-file	Automatically update active configuration parameters of parent model using custom M-file
ERT (optimized for fixed-point)	Automatically update active configuration parameters of parent model for ERT fixed-point code generation
ERT (optimized for floating-point)	Automatically update active configuration parameters of parent model for ERT floating-point code generation

Block	Description
GRT (debug for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation with debugging enabled
GRT (optimized for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation

These are shown in the figure below.


Library: rtweclib/Configuration Wizards
File Edit View Format Help
Configure Model (double-click to activate)
ERT (optimized for fixed-point)
Configure Model (double-click to activate)
ERT (optimized for floating-point)
Configure Model (double-click to activate)
GRT (optimized for fixed/floating-point)
Configure Model (double-click to activate)
GRT (debug for fixed/floating-point)
Configure Model (double-click to activate)
Custom M-file

When you add one of the preset Configuration Wizard blocks to your model and double-click it, a predefined M-file script executes and configures all parameters of the model's active configuration set without manual intervention. The preset blocks configure the options optimally for one of the following cases:

- Fixed-point code generation with the ERT target
- Floating-point code generation with the ERT target
- Fixed/floating-point code generation with TLC debugging options enabled, with the GRT target.
- Fixed/floating-point code generation with the GRT target

The Custom block is associated with an example M-file script that you can adapt to your requirements.

You can also set up the Configuration Wizard blocks to invoke the build process after configuring the model.

Configuration Wizards vs. Auto-Configuring Targets

Configuration Wizard scripts and auto-configuring targets offer two different approaches to automatic model configuration. You need to consider issues of complexity and the needs of your end users when choosing one or the other approach.

Auto-configuring targets (described in "Auto-Configuring Models for Code Generation" on page 5-22 and "Generating Efficient Code with Optimized ERT Targets" on page 5-26) execute a back end configuration function (hook file) during the code generation process. The auto-configuration function in effect bypasses the options set in the model's configuration set, which are saved and restored transparently across the build process.

Configuration Wizards, on the other hand, execute a configuration script independently from the code generation process. The Configuration Wizard script actually changes the model's active configuration set. These changes are then visible in the GUI and can be saved with the model.

It is generally simpler to create a custom Configuration Wizard script than to create a custom auto-configuring target. Creating a Configuration Wizard script, in many cases, requires only simple modifications to an existing template. Creating a custom auto-configuring target, on the other hand, requires some knowledge of the internals of the build process.

Adding a Configuration Wizard Block to Your Model

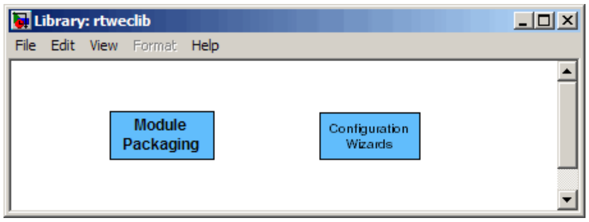
This section describes how to add one of the preset Configuration Wizard blocks to a model.

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

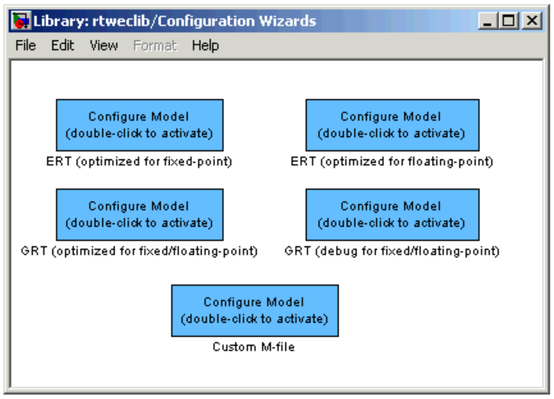
1. Open the model that you want to configure.

2. Open the Real-Time Workshop Embedded Coder library by typing the command `rtweclib`.

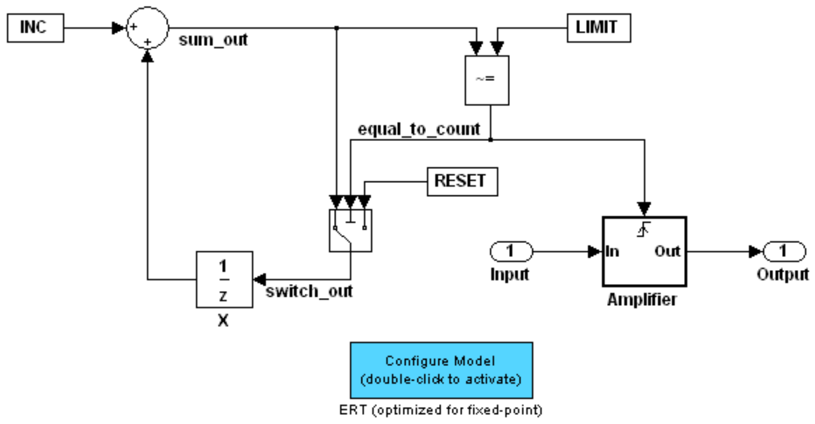
3 The top level of the library is shown below.



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

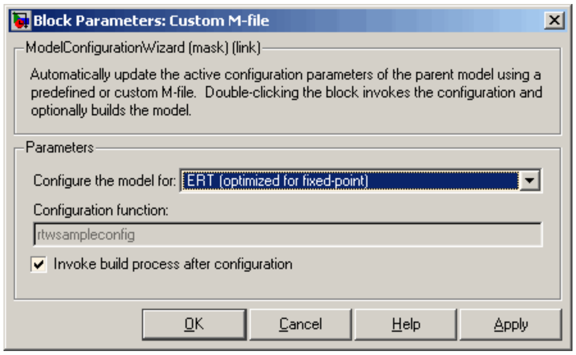

Library: rtweclib/Configuration Wizards
File Edit View Format Help
Configure Model (double-click to activate)
ERT (optimized for fixed-point)
Configure Model (double-click to activate)
ERT (optimized for floating-point)
Configure Model (double-click to activate)
GRT (optimized for fixed/floating-point)
Configure Model (double-click to activate)
GRT (debug for fixed/floating-point)
Configure Model (double-click to activate)
Custom M-file

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



6 You can set up the Configuration Wizard block to invoke the build process after executing its configuration script. If you do not want to use this feature, skip to the next step.

If you want the Configuration Wizard block to invoke the build process, right-click on the Configuration Wizard block in your model, and select **Mask Parameters...** from the context menu. Then, select the **Invoke build process after configuration** option, as shown below.


Block Parameters: Custom M-file

ModelConfigurationWizard (mask) (link)

Automatically update the active configuration parameters of the parent model using a predefined or custom M-file. Double-clicking the block invokes the configuration and optionally builds the model.

Parameters

Configure the model for: ERT (optimized for fixed-point)

Configuration function:

rtwsampleconfig

✓ Invoke build process after configuration

7 Click **Apply**, and close the Mask Parameters dialog box.

Note You should not change the **Configure the model for** option, unless you want to create a custom block and script. In that case, see "Creating a Custom Configuration Wizard Block" on page 5-67.

---

8 Save the model.

9 You can now use the Configuration Wizard block to configure the model, as described in the next section.

Using Configuration Wizard Blocks

Once you have added a Configuration Wizard block to your model, just double-click the block. The script associated with the block automatically sets all parameters of the active configuration set that are relevant to code generation (including selection of the appropriate target). You can verify that the options have changed by opening the Configuration Parameters dialog box and examining the settings.

If the **Invoke build process after configuration** option for the block was selected, the script also initiates the code generation and build process.

Note You can add more than one Configuration Wizard block to your model. This provides a quick way to switch between configurations.

Creating a Custom Configuration Wizard Block

The Custom Configuration Wizard block is shipped with an associated M-file script, `rtwsampleconfig.m`. The script is located in the directory *`matlabroot`*`/toolbox/rtw/rtw`.

Both the block and the script are intended to provide a starting point for customization. This section describes:

- How to create a custom Configuration Wizard block linked to a custom script.

- Operation of the example script, and programming conventions and requirements for a customized script.
- How to run a configuration script from the MATLAB command line (without a block).

Setting Up a Configuration Wizard Block

This section describes how to set up a custom Configuration Wizard block and link it to a script. If you want to use the block in more than one mode, it is advisable to create a Simulink library to contain the block.

To begin, make a copy of the example script for later customization:

1. Create a directory to store your custom script. This directory should not be anywhere inside the MATLAB directory structure (that is, it should not be under `matlabroot`).

   The discussion below refers to this directory as `/my_wizards`.

2. Add the directory to the MATLAB path. Save the path for future sessions.

3. Copy the example script (`matlabroot/toolbox/rtw/rtw/rtwsampleconfig.m`) to the `/my_wizards` directory you created in the previous steps. Then, rename the script as desired. The discussion below uses the name `my_configscript.m`.

4. Open the example script into the MATLAB editor. Scroll to the end of the file and enter the following line of code:

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

   This statement is used later as a test to verify that your custom block has executed the script.

5. Save your script and close the MATLAB editor.

The next step is to create a Simulink library and add a custom block to it. Do this as follows:

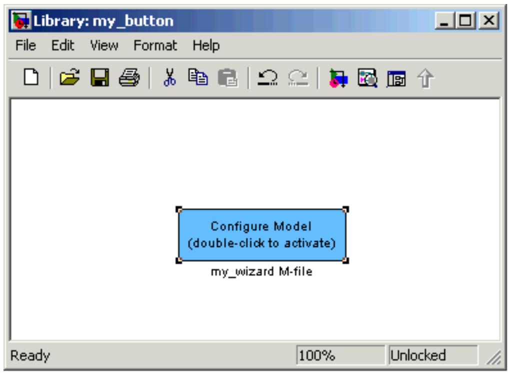
1 Open the Real-Time Workshop Embedded Coder library and the Configuration Wizards sublibrary, as described in "Adding a Configuration Wizard Block to Your Model" on page 5-64.

2 Select **New Library** from the **File** menu of the Configuration Wizards sublibrary window. An empty library window opens.

3 Select the `Custom M-file` block from the Configuration Wizards sublibrary and drag and drop it into the empty library window.

4 To distinguish your custom block from the original, edit the `Custom M-file` label under the block as desired.

5 Select **Save as** from the **File** menu of the new library window; save the library to the `/my_wizards` directory, under your library name of choice. In the figure below, the library has been saved as `my_button`, and the block has been labeled `my_wizard M-file`.

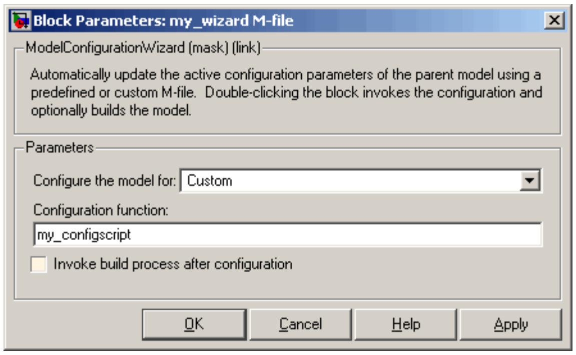


The next step is to link the custom block to the custom script:

1 Right-click on the block in your model, and select **Mask Parameters** from the context menu. Notice that the **Configure the model for** menu set to `Custom`. When `Custom` is selected, the **Configuration function** edit field is enabled, so you can enter the name of a custom script.

2 Enter the name of your custom script into the **Configuration function** field. (Do not enter the `.m` filename extension, which is implicit.) In the figure below, the script name `my_configscript` has been entered into the

Configuration function field. This establishes the linkage between the block and script.


Block Parameters: my_wizard M-file

ModelConfigurationWizard (mask) (link)

Automatically update the active configuration parameters of the parent model using a predefined or custom M-file. Double-clicking the block invokes the configuration and optionally builds the model.

Parameters

Configure the model for: Custom

Configuration function:

my_configscript

☐ Invoke build process after configuration

3 Note that by default, the **Invoke build process after configuration** option is deselected. You can change the default for your custom block by selecting this option. For now, leave this option deselected.

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

5 Save the library.

6 Close the Real-Time Workshop Embedded Coder library and the Configuration Wizards sublibrary. Leave your custom library open for use in the next step.

Now, test your block and script in a model. Do this as follows:

1. Open the `vdp` demo model by typing the command:

   ```
   vdp
   ```

2. Open the Configuration Parameters dialog box and view the Real-Time Workshop options by clicking on the **Real-Time Workshop** entry in the list in the left pane of the dialog box.

3. Observe that the `vdp` demo is configured, by default, for the GRT target. Close the Configuration Parameters dialog box.

4. Select your custom block from your custom library. Drag and drop the block into the `vdp` model.

5. In the `vdp` model, double-click your custom block.

6. In the MATLAB window, you should see the test message you previously added to your script:

   ```
   Custom Configuration Wizard Script completed.
   ```

   This indicates that the custom block successfully executed the script.

7. Reopen the Configuration Parameters dialog box and view the **Real-Time Workshop** options again. You should now see that the model is configured for the ERT target.

Before applying further edits to your custom script, proceed to the next section to learn about the operation and conventions of Configuration Wizard scripts.

Creating a Configuration Wizard Script

You should create your custom Configuration Wizard script by copying and modifying the example script, `rtwsampleconfig.m`. This section provides guidelines for modification.

The Configuration Function. The example script implements a single function without a return value. The function takes a single argument `cs`:

```
function rtwsampleconfig(cs)
```

The argument `cs` is a handle to a proprietary object that contains information about the model's active configuration set. Simulink obtains this handle and passes it in to the configuration function when the user double-clicks a Configuration Wizard block.

Your custom script should conform to this prototype. Your code should use `cs` as a "black box" object that transmits information to and from the active configuration set, using the accessor functions described below.

Accessing Configuration Set Options. To set options or obtain option values, use the Simulink `set_param` and `get_param` functions (if you are unfamiliar with these functions, see the Simulink Reference document).

Option names are passed in to `set_param` and `get_param` as strings specifying an *internal option name*. The internal option name is not always the same as the corresponding option label on the GUI (for example, the Configuration Parameters dialog box). The example configuration accompanies each `set_param` and `get_param` call with a comment that correlates internal option names to GUI option labels. For example:

```
set_param(cs,'LifeSpan','1'); % Application lifespan (days)
```

To obtain the current setting of an option in the active configuration set, call `get_param`. Pass in the `cs` object as the first argument, followed by the internal option name. For example, the following code excerpt tests the setting of the **Generate HTML report** option:

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

To set an option in the active configuration set, call `set_param`. Pass in the `cs` object as the first argument, followed by one or more parameter/value pairs that specify the internal option name and its value. For example, the following code excerpt turns off the **Support absolute time** option:

```
set_param(cs,'SupportAbsoluteTime','off');
```

Selecting a Target. A Configuration Wizard script must select a target configuration. The example script uses the ERT target as a default. The script first stores string variables that correspond to the required **System target file**, **Template makefile**, and **Make command** settings:

```
stf = 'ert.tlc';
tmf = 'ert_default_tmf';
mc  = 'make_rtw';
```

The system target file is selected by passing the `cs` object and the `stf` string to the `switchTarget` function:

```
switchTarget(cs,stf,[]);
```

The template makefile and make command options are set by `set_param` calls:

```
set_param(cs,'TemplateMakefile',tmf);
set_param(cs,'MakeCommand',mc);
```

To select a target, your custom script needs only to set up the string variables `stf`, `tmf`, and `mc` and pass them to the appropriate calls, as above.

Obtaining Target and Configuration Set Information. The following utility functions and properties are provided so that your code can obtain information about the current target and configuration set, with the `cs` object:

- `isValidParam(cs, 'option')`: The `option` argument is an internal option name. `isValidParam` returns true if `option` is a valid option in the context of the active configuration set.
- `getPropEnabled(cs, 'option')`: The `option` argument is an internal option name. Returns true if this `option` is enabled (that is, writable).
- `IsERTTarget` property: Your code can detect whether or not the currently selected target is derived from the ERT target is selected by checking the `IsERTTarget` property, as follows:

```
    isERT = strcmp(get_param(cs,'IsERTTarget'),'on');
```

This information can be used to determine whether or not the script should configure ERT-specific options, for example:

```
if isERT
  set_param(cs,'ZeroExternalMemoryAtStartup','off');
  set_param(cs,'ZeroInternalMemoryAtStartup','off');
  set_param(cs,'InitFltsAndDblsToZero','off');
  set_param(cs,'InlinedParameterPlacement',...
                'NonHierarchical');
     set_param(cs,'NoFixptDivByZeroProtection','on')
end
```

Invoking a Configuration Wizard Script from the MATLAB Command Prompt

Like any other M-file, Configuration Wizard scripts can be run from the MATLAB command prompt. (The Configuration Wizard blocks are provided as a graphical convenience, but are not essential.)

Before invoking the script, you must open a model and instantiate a `cs` object to pass in as an argument to the script. After running the script, you can invoke the build process with the `rtwbuild` command. The following example opens, configures, and builds a model.

```
open my_model;
cs = getActiveConfigSet ('my_model');
rtwsampleconfig(cs);
rtwbuild('my_model');
```

Replacement of STF_rtw_info_hook Mechanism

Prior to MATLAB Release 14, custom targets supplied target-specific information with a hook file (referred to as `STF_rtw_info_hook.m`). The `STF_rtw_info_hook` specified properties such as word sizes for integer data types (for example, `char`, `short`, `int`, and `long`), and C implementation-specific properties of the custom target.

The `STF_rtw_info_hook` mechanism has been replaced by the **Hardware Implementation** pane of the Configuration Parameters dialog box. Using this dialog box, you can specify all properties that were formerly specified in your `STF_rtw_info_hook` file.

For backward compatibility, existing `STF_rtw_info_hook` files continue to operate correctly. However, you should convert your target and models to use of the **Hardware Implementation** pane. See the “Configuring Hardware Properties and Constraints” section of the Real-Time Workshop documentation.

Optimizing Task Scheduling for Multirate Multitasking Models on RTOS Targets

In this section...

Overview

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

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

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

The event flags indicate whether or not a given task is scheduled for execution. For a multirate, multitasking model, the event flags are maintained by the *`model`*`_SetEventsForThisBaseStep` function. *`model`*`_SetEventsForThisBaseStep` invokes the macro `rtmStepTask` to test the value of each counter. `rtmStepTask` returns `TRUE` when a counter indicates that a task's sample period has elapsed. When this occurs, `*model*_SetEventsForThisBaseStep` sets the event flag for that task.

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

For bareboard targets that cannot rely on an external RTOS, the event flags are mandatory to allow overlapping task preemption. However, an RTOS target uses the operating system itself to manage overlapping task preemption, making the maintenance of the event flags redundant. An RTOS target can eliminate the call to `*model*_SetEventsForThisBaseStep`, and examine the counters by invoking `rtmStepTask` directly.

Using rtmStepTask

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

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

The arguments are:

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

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

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

Task Scheduling Code for Multirate Multitasking Model on VxWorks Target

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

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

Suppressing Redundant Scheduling Calls

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

```
%assign SuppressSetEventsForThisBaseRateFcn = 1
```

Target Function Libraries

Introduction to Target Function Libraries

In this section...
“Overview” on page 6-2
“TFL General Workflow” on page 6-5
“TFL Quick-Start Example” on page 6-6

Overview

Real-Time Workshop Embedded Coder provides the target function library (TFL) API, which allows you to create and register function replacement tables. When selected for a model, these TFL tables provide the basis for replacing default math functions and operators in your model code with target-specific code. The ability to control function and operator replacements in this manner potentially allows you to optimize target performance (speed and memory) and better integrate model code with external and legacy code.

A *target function library* (TFL) is a set of one or more function replacement tables that define the target-specific implementations of math functions and operators to be used in generating code for your Simulink model. Real-Time Workshop provides three default TFLs, which you can select from the **Target function library** drop-down list on the **Interface** pane of the Configuration Parameters dialog box.

TFL	Description	Contains tables...
`C89/C90 (ANSI)`	Generates calls to the ISO/IEC 9899:1990 C standard math library for floating-point functions.	`ansi_tfl_table_tmw.mat`
`C99 (ISO)`	Generates calls to the ISO/IEC 9899:1999 C standard math library.	`iso_tfl_table_tmw.mat` `ansi_tfl_table_tmw.mat`

`GNU99 (GNU)`	Generates calls to the GNU gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.	`gnu_tfl_table_tmw.mat`
		`iso_tfl_table_tmw.mat`
		`ansi_tfl_table_tmw.mat`

When a TFL contains multiple tables, the order in which they are listed reflects the order in which they are searched. The TFL API allows you to create your own TFLs, made up of your own function tables in combination with one of the three default TFLs. For example, you could create a TFL for an embedded processor that combines some special-purpose function customizations with a processor-specific library of function and operator implementations:

`MyProcessor (ANSI)`	Generates calls to my custom function implementations or a processor-specific library.	`tfl_table_sinfcns.m` `tfl_table_myprocessor.m` `ansi_tfl_table_tmw.mat`

Each TFL function replacement table contains one or more table entries, with each table entry representing a potential replacement for a single math function or an operator. Each table entry provides a mapping between a *conceptual view* of the function or operator (similar to the Simulink block view of the function or operator) and a *target-specific implementation* of that function or operator.

The conceptual view of a function or operator is represented in a TFL table entry by the following elements, which identify the function or operator entry to the code generation process:

- A function or operator key (a function name such as `'cos'` or an operator ID string such as `'RTW_OP_ADD'`)
- A set of conceptual arguments that observe Simulink-based naming (`'y1'`, `'u1'`, `'u2'`, ...), along with their I/O types (output or input) and data types
- Other attributes, such as fixed-point saturation and rounding characteristics for operators, as needed to identify the function or operator to the code generation process as exactly as you require for matching purposes

The target-specific implementation of a function or operator is represented in a TFL table entry by the following elements:

- The name of your implementation function (such as `'cos_dbl'` or `'u8_add_u8_u8'`)
- A set of implementation arguments that you define (the order of which must correspond to the conceptual arguments), along with their I/O types (output or input) and data types

- Parameters providing the build information for your implementation function, including header file and source file names and paths

Additionally, a TFL table entry includes a priority value (0-100, with 0 as the highest priority), which defines the entry's priority relative to other entries in the table.

During code generation for your model, when the code generation process encounters a call site for a math function or operator, it creates and partially populates a TFL entry object, for the purpose of querying the TFL database for a replacement function. The information provided for the TFL query includes the function or operator key and the conceptual argument list. The TFL entry object is then passed to the TFL and, if there is a matching table entry in the TFL, a fully-populated TFL entry, including the implementation function name, argument list, and build information, is returned to the call site and used to generate code.

Within the TFL that is selected for your model, the tables that comprise the TFL are searched in the order in which they are listed (by `RTW.viewTFL` or by the TFL's **Target function library** tool tip). Within each table, if multiple matches are found for a TFL entry object, priority level determines the match that is returned. A higher-priority (lower-numbered) entry will be used over a similar entry with a lower priority (higher number).

The TFL API supports the following math functions for replacement with custom library functions using TFL tables:

`abs`	`ceil`	`floor`	`sinh`
`acos`	`cos`	`log`	`sqrt`
`asin`	`cosh`	`log10`	`tan`
`atan`	`exp`	`sin`	`tanh`

The TFL API also supports the following scalar operators for replacement with custom library functions using TFL tables:

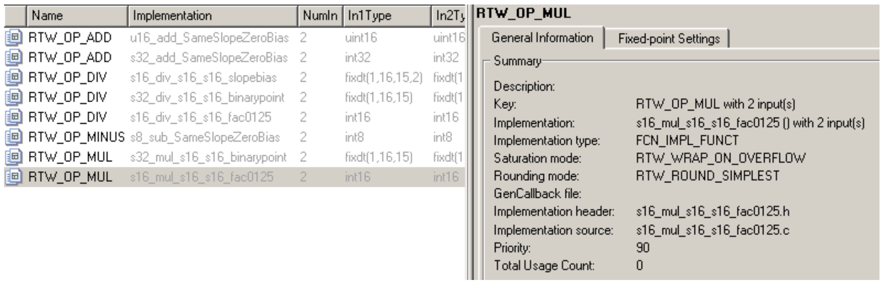
`+` (addition)
`-` (subtraction)
`*` (multiplication)

/ (division)

TFL General Workflow

The general steps for creating and using a target function library are as follows:

1. Create one or more TFL tables containing replacement entries for math operators (+, –, *, /) and functions using a MATLAB-based API. (The demo `rtwdemo_tfl_script` provides example tables that can be used as a starting point for customization.)



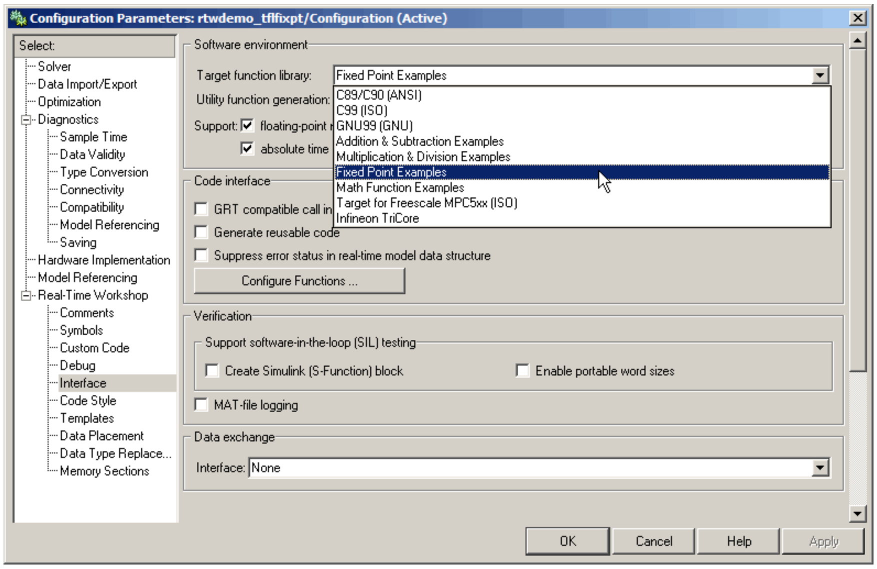
Name	Implementation	NumIn	In1Type	In2Ty
RTW_OP_ADD	u16_add_SameSlopeZeroBias	2	uint16	uint16
RTW_OP_ADD	s32_add_SameSlopeZeroBias	2	int32	int32
RTW_OP_DIV	s16_div_s16_s16_slopebias	2	fixdt(1,16,15,2)	fixdt(1
RTW_OP_DIV	s32_div_s16_s16_binarypoint	2	fixdt(1,16,15)	fixdt(1
RTW_OP_DIV	s16_div_s16_s16_fac0125	2	int16	int16
RTW_OP_MINUS	s8_sub_SameSlopeZeroBias	2	int8	int8
RTW_OP_MUL	s32_mul_s16_s16_binarypoint	2	fixdt(1,16,15)	fixdt(1
RTW_OP_MUL	s16_mul_s16_s16_fac0125	2	int16	int16

RTW_OP_MUL

Description:	
Key:	RTW_OP_MUL with 2 input(s)
Implementation:	s16_mul_s16_s16_fac0125 () with 2 input(s)
Implementation type:	FCN_IMPL_FUNCT
Saturation mode:	RTW_WRAP_ON_OVERFLOW
Rounding mode:	RTW_ROUND_SIMPLEST
GenCallback file:	
Implementation header:	s16_mul_s16_s16_fac0125.h
Implementation source:	s16_mul_s16_s16_fac0125.c
Priority:	90
Total Usage Count:	0

2. Register a target function library, consisting of one or more replacement tables, using a Simulink `sl_customization` API.

3. Open your model and select the desired target function library from the **Target function library** drop-down list, located on the **Interface** pane in the Configuration Parameters dialog box.



Configuration Parameters: rtwdemo_tflfixpt/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
  - Sample Time
  - Data Validity
  - Type Conversion
  - Connectivity
  - Compatibility
  - Model Referencing
  - Saving
- Hardware Implementation
- Model Referencing
- Real-Time Workshop
  - Comments
  - Symbols
  - Custom Code
  - Debug
  - Interface
  - Code Style
  - Templates
  - Data Placement
  - Data Type Replace...
  - Memory Sections

Software environment

Target function library: Fixed Point Examples

Utility function generation:

Support: ☑ floating-point ☑ absolute time

C89/C90 (ANSI)
C99 (ISO)
GNU99 (GNU)
Addition & Subtraction Examples
Multiplication & Division Examples
Fixed Point Examples
Math Function Examples
Target for Freescale MPC5xx (ISO)
Infineon TriCore

Code interface

☐ GRT compatible call in

☐ Generate reusable code

☐ Suppress error status in real-time model data structure

Configure Functions ...

Verification

Support software-in-the-loop (SIL) testing

☐ Create Simulink (S-Function) block ☐ Enable portable word sizes

☐ MAT-file logging

Data exchange

Interface: None

OK Cancel Help Apply

4 Build your model.

See the demo `rtwdemo_tfl_script`, which illustrates how to use TFLs to replace operators and functions in generated code. With each example model included in this demo, a separate TFL is provided to illustrate the creation of operator and function replacements and how to register the replacements with Simulink.

TFL Quick-Start Example

This section walks you through a simple example of the complete TFL workflow. (The materials for this example can easily be created based on the file and model displays in this section.)

1 Create and save a TFL table definition file that instances and populates a TFL table entry, such as the file `tfl_table_sinfcn.m` shown below. This file creates function table entries for the `sin` function. For detailed information on creating table definition files for math functions and operators, see "Creating Function Replacement Tables" on page 6-13.

```
function hTable = tfl_table_sinfcn()
%TFL_TABLE_SINFCN - Describe function entries for a Target Function Library table.
```

```
hTable = RTW.TflTable;

% Create entry for double data type sine function replacement
hTable.registerCFunctionEntry(100, 1, 'sin', 'double', 'sin_dbl', ...
                                             'double', '<sin_dbl.h>','','');

% Create entry for single data type sine function replacement
hTable.registerCFunctionEntry(100, 1, 'sin', 'single', 'sin_sgl', ...
                                             'double', '<sin_sgl.h>','','');
```

Note See "Example: Mapping Math Functions to Target-Specific Implementations" on page 6-22 for another example of `sin` function replacement, in which function arguments are created individually.

---

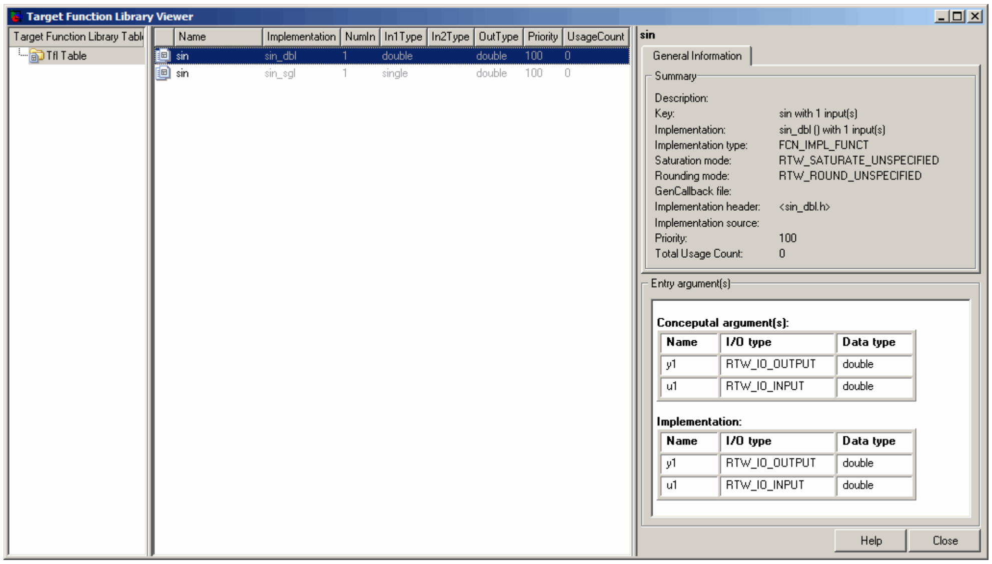
2 As a first check of the validity of your table entries, invoke the TFL table definition file as follows:

```
>> tbl = tfl_table_sinfcn

tbl =

RTW.TflTable
            Version: '1.0'
         AllEntries: [2x1 RTW.TflCFunctionEntry]
    ReservedSymbols: []

>>
```

Any errors found during the invocation will be displayed.

3 As a further check of your table entries, invoke the TFL Viewer using the following MATLAB command:

```
>> RTW.viewTfl(tfl_table_sinfcn)
```



Name	Implementation	NumIn	In1Type	In2Type	OutType	Priority	UsageCount
sin	sin_dbl	1	double		double	100	0
sin	sin_sgl	1	single		double	100	0

sin

Summary

Description:	
Key:	sin with 1 input(s)
Implementation:	sin_dbl () with 1 input(s)
Implementation type:	FCN_IMPL_FUNCT
Saturation mode:	RTW_SATURATE_UNSPECIFIED
Rounding mode:	RTW_ROUND_UNSPECIFIED
GenCallback file:	
Implementation header:	<sin_dbl.h>
Implementation source:	
Priority:	100
Total Usage Count:	0

Entry argument(s)

Conceptual argument(s):

Name	I/O type	Data type
y1	RTW_IO_OUTPUT	double
u1	RTW_IO_INPUT	double

Implementation:

Name	I/O type	Data type
y1	RTW_IO_OUTPUT	double
u1	RTW_IO_INPUT	double

Select entries in your table and verify that the graphical display of the contents of your table meets your expectations. (The TFL Viewer can also help you debug issues with the order of entries in a table, the order of tables in a TFL, and function signature mismatches. For more information, see "Examining and Validating Function Replacement Tables" on page 6-58.)

4 Create and save a TFL registration file that includes the `tfl_table_sinfcn` table, such as the `sl_customization.m` file shown below. The file specifies that the TFL to be registered is named `'Sine Function Example'` and consists of `tfl_table_sinfcn`, with the default ANSI math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION


% Local function to define a TFL containing tfl_table_sinfcn
```

```
function thisTfl = locTflRegFcn

  % Instance a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Sine Function Example';
  thisTfl.Description = 'Demonstration of sine function replacement';
  thisTfl.TableList = {'tfl_table_sinfcn'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

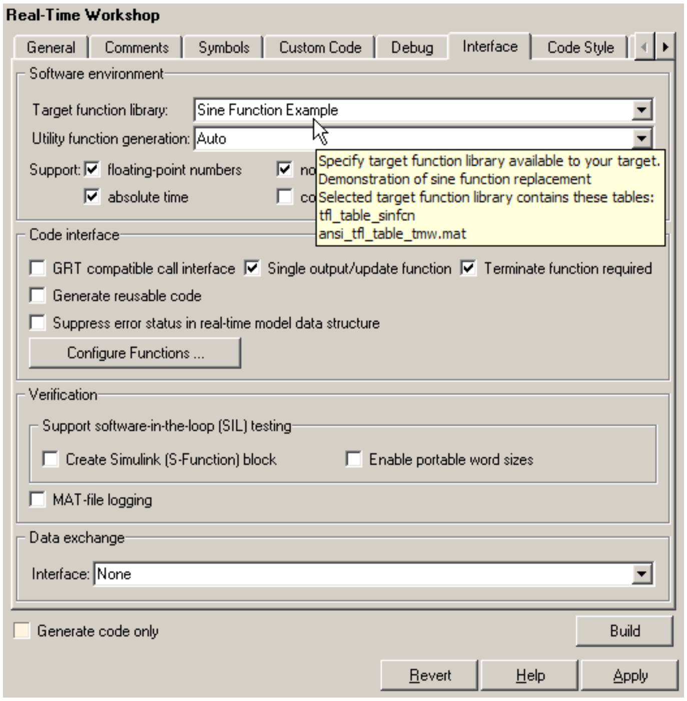
If you place this `sl_customization.m` file in the MATLAB search path or in the current working directory, the TFL will be registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`.

For more information about using a TFL registration file, see "Registering Target Function Libraries" on page 6-66.

5 With your `sl_customization.m` file in the MATLAB search path or in the current working directory, open an ERT-based model in Simulink and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip is displayed containing information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Real-Time Workshop

General | Comments | Symbols | Custom Code | Debug | Interface | Code Style

Software environment

Target function library: Sine Function Example

Utility function generation: Auto

Support: ☑ floating-point numbers ☑ no ☑ absolute time ☐ co

Specify target function library available to your target.
Demonstration of sine function replacement
Selected target function library contains these tables:
tfl_table_sinfcn
ansi_tfl_table_tmw.mat

Code interface

☐ GRT compatible call interface ☑ Single output/update function ☑ Terminate function required

☐ Generate reusable code

☐ Suppress error status in real-time model data structure

Configure Functions ...

Verification

Support software-in-the-loop (SIL) testing

☐ Create Simulink (S-Function) block ☐ Enable portable word sizes

☐ MAT-file logging

Data exchange

Interface: None

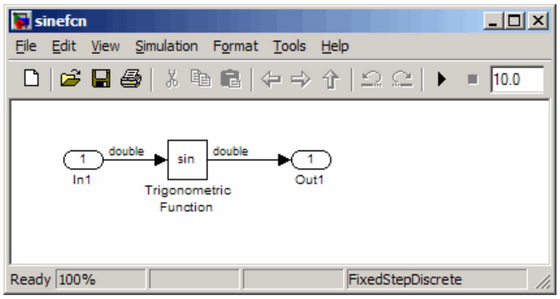
☐ Generate code only

Build

Revert Help Apply

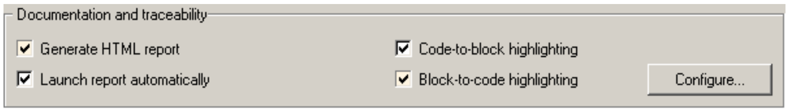
Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Sine Function Example`.

6 Create an ERT-based model with a Trigonometric Function block set to the sine function, such as the following:



Make sure that the TFL you registered, `Sine Function Example`, is selected for this model.

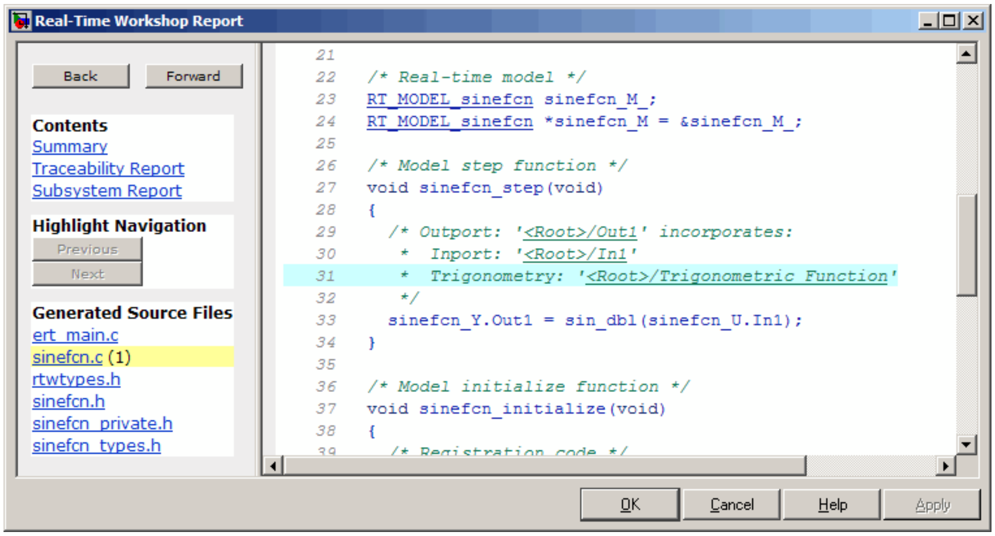
7 Go to the **Real-Time Workshop** pane of the Configuration Parameters dialog box and select the options **Generate HTML report** and **Block-to-code highlighting**. Select the **Generate code only** option and generate code for the model.


Documentation and traceability

☑ Generate HTML report ☑ Code-to-block highlighting

☑ Launch report automatically ☑ Block-to-code highlighting

8 Go to the model window and use block-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the Trigonometric Function block and select **Real-Time Workshop > Highlight Code**. This highlights the `sin` function code within the model step function in `sinefcn.c`. In this case, `sin` has been replaced with `sin_dbl` in the generated code.



```

/* Real-time model */
RT_MODEL_sinefcn sinefcn_M_;
RT_MODEL_sinefcn *sinefcn_M = &sinefcn_M_;

/* Model step function */
void sinefcn_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Trigonometry: '<Root>/Trigonometric Function'
   */
  sinefcn_Y.Out1 = sin_dbl(sinefcn_U.In1);
}

/* Model initialize function */
void sinefcn_initialize(void)
{
  /* Registration code */
```

9 If function replacements did not happen as you intended, you can use the techniques described in "Examining and Validating Function Replacement Tables" on page 6-58 to help you determine why the code generation process was unable to match a function signature with the TFL table entry you created for it.

For example, you can view the TFL cache hits and misses logged during the most recent build. For the code generation step in this example, there was one cache hit and zero cache misses, as shown in the `HitCache` and `MissCache` entries displayed below.

```
>> a=get_param('sinefcn','TargetFcnLibHandle')

a =

RTW.TflControl
        Version: '1.0'
       HitCache: [1x1 RTW.TflCFunctionEntry]
      MissCache: [0x1 handle]
    TLCCallList: [0x1 handle]
      TflTables: [2x1 RTW.TflTable]

>> a.HitCache(1)

ans =

RTW.TflCFunctionEntry
                        Key: 'sin'
                   Priority: 100
             ConceptualArgs: [2x1 RTW.TflArgNumeric]
             Implementation: [1x1 RTW.CImplementation]
.
.
.
>>
```

Creating Function Replacement Tables

In this section...

Overview

To create a TFL table containing replacement information for supported functions and operators, you perform the following steps:

1. Create a table definition M-file containing a function definition in the following general form:

   ```
   function hTable = tfl_table_name()
   %TFL_TABLE_NAME - Describe entries for a Target Function Library table.
   .
   .
   .
   ```

2. Within the function body, instance a TFL table with a command such as the following:

   ```
   hTable = RTW.TflTable;
   ```

3. Use the TFL table creation functions (listed in the table below) to add table entries representing your replacements for supported functions and operators. For each individual function or operator entry, you issue one or more function calls to

a Instance a table entry.

b Add conceptual arguments, implementation arguments, and other attributes to the entry.

c Add the entry to the table.

"Creating Table Entries" on page 6-16 describes this procedure in detail, including two methods for creating function entries. Here is a sample function entry from the "TFL Quick-Start Example" on page 6-6:

```
% Create entry for double data type sine function replacement
hTable.registerCFunctionEntry(100, 1, 'sin', 'double', 'sin_dbl', ...
                                             'double', '<sin_dbl.h>','','');
```

4 Save the table definition M-file using the name of the table definition function, for example, `tfl_table_sinfcn.m`.

After you have created a table definition M-file, you can do the following:

- Examine and validate the table, as described in "Examining and Validating Function Replacement Tables" on page 6-58
- Register a TFL containing the table with Simulink, as described in "Registering Target Function Libraries" on page 6-66

Once a TFL is registered with Simulink, it is displayed in the Simulink GUI and can be selected for use in building models.

The following table provides a functional grouping of the TFL table creation functions.

Function	Description
Table entry creation	
`addEntry`	Add table entry to collection of table entries registered in TFL table
`copyConceptualArgsToImplementation`	Copy conceptual argument specifications to matching implementation arguments for TFL table entry
`createAndAddConceptualArg`	Create conceptual argument from specified properties and add to conceptual arguments for TFL table entry

Function	Description
`createAndAddImplementationArg`	Create implementation argument from specified properties and add to implementation arguments for TFL table entry
`createAndSetCImplementationReturn`	Create implementation return argument from specified properties and add to implementation for TFL table entry
`setTflCFunctionEntryParameters`	Set specified parameters for function entry in TFL table
`setTflCOperationEntryParameters`	Set specified parameters for operator entry in TFL table
Alternative method for conceptual argument creation	
`addConceptualArg`	Add conceptual argument to array of conceptual arguments for TFL table entry
`getTflArgFromString`	Create TFL argument based on specified name and built-in data type
Alternative method for function entry creation	
`registerCFunctionEntry`	Create TFL function entry based on specified parameters and register in TFL table
`registerCPromotableMacroEntry`	Create TFL promotable macro entry based on specified parameters and register in TFL table (for `abs` function replacement only)
Build information	
`addAdditionalHeaderFile`	Add additional header file to array of additional header files for TFL table entry
`addAdditionalIncludePath`	Add additional include path to array of additional include paths for TFL table entry
`addAdditionalLinkObj`	Add additional link object to array of additional link objects for TFL table entry
`addAdditionalLinkObjPath`	Add additional link object path to array of additional link object paths for TFL table entry

Function	Description
`addAdditionalSourceFile`	Add additional source file to array of additional source files for TFL table entry
`addAdditionalSourcePath`	Add additional source path to array of additional source paths for TFL table entry
Reserved identifiers	
`setReservedIdentifiers`	Register specified reserved identifiers to be associated with TFL table

Creating Table Entries

- "Introduction" on page 6-16
- "General Method for Creating Function and Operator Entries" on page 6-18
- "Alternative Method for Creating Function Entries" on page 6-21

Introduction

You define TFL table entries by issuing TFL table creation function calls inside a table definition M-file. The function calls must follow a function declaration and a TFL table instantiation, such as the following:

```
function hTable = tfl_table_sinfcn()
%TFL_TABLE_SINFCN - Describe function entries for a Target Function Library table.

hTable = RTW.TflTable;
```

Within the function body, you use the TFL table creation functions to add table entries representing your replacements for supported functions and operators. For each individual function or operator entry, you issue one or more function calls to

1. Instance a table entry.
2. Add conceptual arguments, implementation arguments, and other attributes to the entry.
3. Add the entry to the table.

The general method for creating function and operator entries, described in "General Method for Creating Function and Operator Entries" on page 6-18, uses the following functions:

Function	Description
Table entry creation	
`addEntry`	Add table entry to collection of table entries registered in TFL table
`copyConceptualArgsToImplementation`	Copy conceptual argument specifications to matching implementation arguments for TFL table entry
`createAndAddConceptualArg`	Create conceptual argument from specified properties and add to conceptual arguments for TFL table entry
`createAndAddImplementationArg`	Create implementation argument from specified properties and add to implementation arguments for TFL table entry
`createAndSetCImplementationReturn`	Create implementation return argument from specified properties and add to implementation for TFL table entry
`setTflCFunctionEntryParameters`	Set specified parameters for function entry in TFL table
`setTflCOperationEntryParameters`	Set specified parameters for operator entry in TFL table
Alternative method for conceptual argument creation	
`addConceptualArg`	Add conceptual argument to array of conceptual arguments for TFL table entry
`getTflArgFromString`	Create TFL argument based on specified name and built-in data type

A simpler alternative creation method is available for function entries, with the constraints that input types must be uniform and implementation arguments must use default Simulink naming. The alternative method uses the following functions and is described in "Alternative Method for Creating Function Entries" on page 6-21:

Function	Description
Alternative method for function entry creation	
`registerCFunctionEntry`	Create TFL function entry based on specified parameters and register in TFL table
`registerCPromotableMacroEntry`	Create TFL promotable macro entry based on specified parameters and register in TFL table (for `abs` function replacement only)

General Method for Creating Function and Operator Entries

The general workflow for creating TFL table entries applies equally to function and operator replacements, and involves the following steps.

Note Implementation argument order must match the conceptual argument order. (Remapping the argument order in your implementation is not supported.)

For function entries, if your implementations additionally meet the requirements that all input arguments are of the same type and your implementation arguments use default Simulink naming (return argument `y1` and input arguments `u`*`n`*), you can use a simpler alternative method for creating the entries, as described in "Alternative Method for Creating Function Entries" on page 6-21

---

1 Instantiate a TFL table entry for a function or operator, using one of the following:

   - `fcn_entry = RTW.TflCFunctionEntry;`
   - `op_entry = RTW.TflCOperationEntry;`
   - `op_entry = RTW.TflCOperationEntryGenerator;`

   (`RTW.TflCOperationEntryGenerator` provides advanced fixed-point parameters, described in "Mapping Fixed-Point Operators to Target-Specific Implementations" on page 6-33, that are not available in `RTW.TflCOperationEntry`.)

2 Set the table entry parameters, which are passed in parameter/value pairs to one of the following functions:

- `setTflCFunctionEntryParameters`
- `setTflCOperationEntryParameters`

For example:

```
setTflCFunctionEntryParameters(fcn_entry, ...
                               'Key',                   'sin', ...
                               'Priority',              30, ...
                               'ImplementationName',    'mySin', ...
                               'ImplementationHeaderFile', 'basicMath.h',...
                               'ImplementationSourceFile', 'basicMath.c');
```

For detailed descriptions of the settable function and operator attributes, see the `setTflCFunctionEntryParameters` and `setTflCOperationEntryParameters` reference pages in the Real-Time Workshop Embedded Coder documentation.

3 Create and add conceptual arguments to the function or operator entry. Output arguments must precede input arguments, and the function signature (including argument naming, order, and attributes) must fulfill the signature match sought by function or operator callers. Conceptual argument names follow the default Simulink naming convention:

- For return argument, `y1`
- For input argument names, `u1`, `u2`, ..., `u`*`n`*

You can create and add conceptual arguments in either of two ways:

- Call the `createAndAddConceptualArg` function to create the argument and add it to the table entry. For example:

```
createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
                          'Name',         'y1',...
                          'IOType',       'RTW_IO_OUTPUT',...
                          'DataTypeMode', 'double');
```

- Call the `getTflArgFromString` function to create an argument based on a built-in data type, and then call the `addConceptualArg` function to add the argument to the table entry.

Note If you use `getTflArgFromString`, the `IOType` property of the created argument defaults to `'RTW_IO_INPUT'`, indicating an input argument. For an output argument, you must change the `IOType` value to `'RTW_IO_OUTPUT'` by directly assigning the argument property. See the example below.

```
arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
```

4. Create and add implementation arguments, representing the signature of your implementation function, to the function or operator entry. The implementation argument order must match the conceptual argument order. You can create and add implementation arguments in either of two ways:

   - Call the `copyConceptualArgsToImplementation` function to populate all of the implementation arguments as copies of the previously-created conceptual arguments. For example:

     ```
     copyConceptualArgsToImplementation(fcn_entry);
     ```

   - Call the `createAndSetCImplementationReturn` function to create the implementation return argument and add it to the table entry, and then call the `createAndAddImplementationArg` function to individually create and add each of your implementation arguments. This method allows you to vary argument attributes, including argument naming, as long as conceptual argument order is maintained. For example:

     ```
     createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                                       'Name',       'y1', ...
                                       'IOType',     'RTW_IO_OUTPUT', ...
                                       'IsSigned',   true, ...
                                       'WordLength', 32, ...
                                       'FractionLength', 0);

     createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                                   'Name',       'u1', ...
                                   'IOType',     'RTW_IO_INPUT',...
     ```

```
                                          'IsSigned',   true,...
                                          'WordLength', 32, ...
                                          'FractionLength', 0 );

    createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                                          'Name',       'u2', ...
                                          'IOType',     'RTW_IO_INPUT',...
                                          'IsSigned',   true,...
                                          'WordLength', 32, ...
                                          'FractionLength', 0 );
```

5 Add the function or operator entry to the TFL table using the `addEntry` function. For example:

```
addEntry(hTable, fcn_entry);
```

For complete examples of function entries and operator entries created using the general method, see "Example: Mapping Math Functions to Target-Specific Implementations" on page 6-22 and "Example: Mapping Operators to Target-Specific Implementations" on page 6-28. For syntax examples, see the examples in the TFL table creation function reference pages in the Real-Time Workshop Embedded Coder documentation.

Alternative Method for Creating Function Entries

You can use a simpler alternative method for creating TFL function entries if your function implementation meets the following criteria:

- As with the general method described in "General Method for Creating Function and Operator Entries" on page 6-18, the implementation argument order must match the conceptual argument order.
- All input arguments are of the same type.
- The return argument name and all input argument names follow the default Simulink naming convention:
  - For the return argument, `y1`
  - For input argument names, `u1`, `u2`, ..., `u`*n*

The alternative method for creating function entries involves a single step. Call one of the following functions to create and add conceptual and implementation arguments and register the function entry:

- `registerCFunctionEntry`
- `registerCPromotableMacroEntry` (use only for the `abs` function)

For example:

```
hTable = RTW.TflTable;

registerCFunctionEntry(hTable, 100, 1, 'sqrt', 'double', ...
                       'sqrt', 'double', '<math.h>', '', '');
```

For detailed descriptions of the function arguments, see the `registerCFunctionEntry` and `registerCPromotableMacroEntry` reference pages in the Real-Time Workshop Embedded Coder documentation.

Example: Mapping Math Functions to Target-Specific Implementations

Real-Time Workshop Embedded Coder supports the following math functions for replacement with custom library functions using target function library (TFL) tables:

abs	ceil	floor	sinh
acos	cos	log	sqrt
asin	cosh	log10	tan
atan	exp	sin	tanh

This example uses the method described in "General Method for Creating Function and Operator Entries" on page 6-18 to create a TFL table entry for the `sin` function.

Note See "TFL Quick-Start Example" on page 6-6 for another example of `sin` function replacement, in which function arguments are created using the simpler method described in "Alternative Method for Creating Function Entries" on page 6-21.

---

1 Create and save the following TFL table definition file, `tfl_table_sinfcn2.m`. This file defines a TFL table containing a function replacement entry for the `sin` function. The function body sets selected sine function entry parameters, creates the `y1` and `u1` conceptual arguments individually, and then copies the conceptual arguments to the implementation arguments. Finally the function entry is added to the table.

```
function hTable = tfl_table_sinfcn2()
%TFL_TABLE_SINFCN2 - Describe function entry for a Target Function Library table.

hTable = RTW.TflTable;

% Create entry for sine function replacement
fcn_entry = RTW.TflCFunctionEntry;
setTflCFunctionEntryParameters(fcn_entry, ...
                               'Key',                'sin', ...
                               'Priority',           30, ...
                               'ImplementationName',  'mySin', ...
                               'ImplementationHeaderFile', 'basicMath.h',...
                               'ImplementationSourceFile', 'basicMath.c');

createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
                          'Name',         'y1',...
                          'IOType',       'RTW_IO_OUTPUT',...
                          'DataTypeMode', 'double');

createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
                          'Name',         'u1', ...
                          'IOType',       'RTW_IO_INPUT',...
                          'DataTypeMode', 'double');

copyConceptualArgsToImplementation(fcn_entry);
```

```
addEntry(hTable, fcn_entry);
```

2 Optionally, perform a quick check of the validity of the sine function entry by invoking the table definition file at the MATLAB command line (`>> tbl = tfl_table_sinfcn2`) and by viewing it in the TFL Viewer (`>> RTW.viewTfl(tfl_table_sinfcn2)`). For more information about validating TFL tables, see "Examining and Validating Function Replacement Tables" on page 6-58.

3 Create and save the following TFL registration file, which references the `tfl_table_sinfcn2` table. The file specifies that the TFL to be registered is named `'Sine Function Example 2'` and consists of `tfl_table_sinfcn2`, with the default ANSI math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION


% Local function to define a TFL containing tfl_table_sinfcn2
function thisTfl = locTflRegFcn

  % Instance a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Sine Function Example 2';
  thisTfl.Description = 'Demonstration of sine function replacement';
  thisTfl.TableList = {'tfl_table_sinfcn2'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

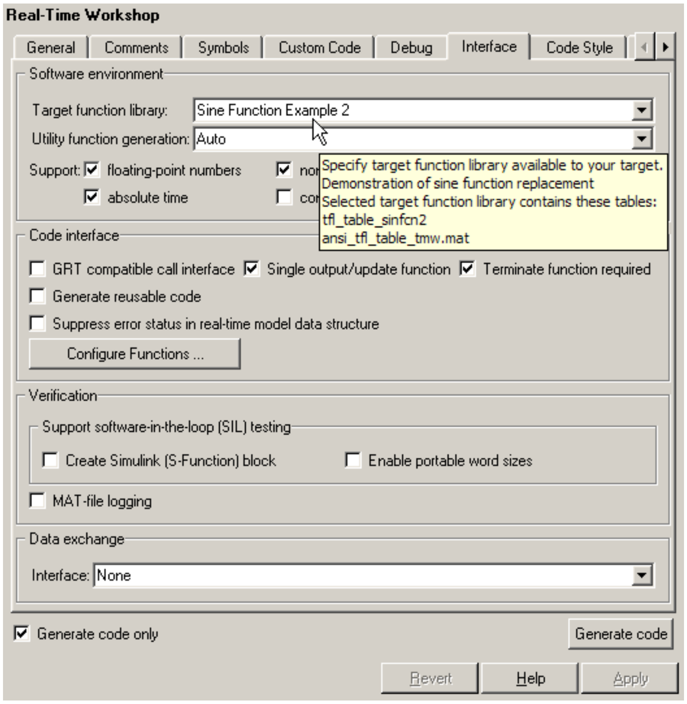
Place this `sl_customization.m` file in the MATLAB search path or in the current working directory, so that the TFL will be registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`.

For more information about using a TFL registration file, see “Registering Target Function Libraries” on page 6-66.

4 With your `sl_customization.m` file in the MATLAB search path or in the current working directory, open an ERT-based model in Simulink and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip is displayed containing information derived from your TFL registration file, such as the TFL description and the list of tables it contains.


Real-Time Workshop

General | Comments | Symbols | Custom Code | Debug | Interface | Code Style

Software environment

Target function library: Sine Function Example 2

Utility function generation: Auto

Support: ☑ floating-point numbers ☑ absolute time

Specify target function library available to your target.
Demonstration of sine function replacement
Selected target function library contains these tables:
tfl_table_sinfcn2
ansi_tfl_table_tmw.mat

Code interface

☐ GRT compatible call interface ☑ Single output/update function ☑ Terminate function required

☐ Generate reusable code

☐ Suppress error status in real-time model data structure

Configure Functions ...

Verification

Support software-in-the-loop (SIL) testing

☐ Create Simulink (S-Function) block ☐ Enable portable word sizes

☐ MAT-file logging

Data exchange

Interface: None

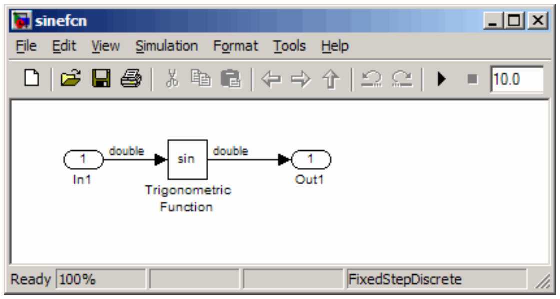
☑ Generate code only

Generate code

Revert Help Apply

Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Sine Function Example 2`.

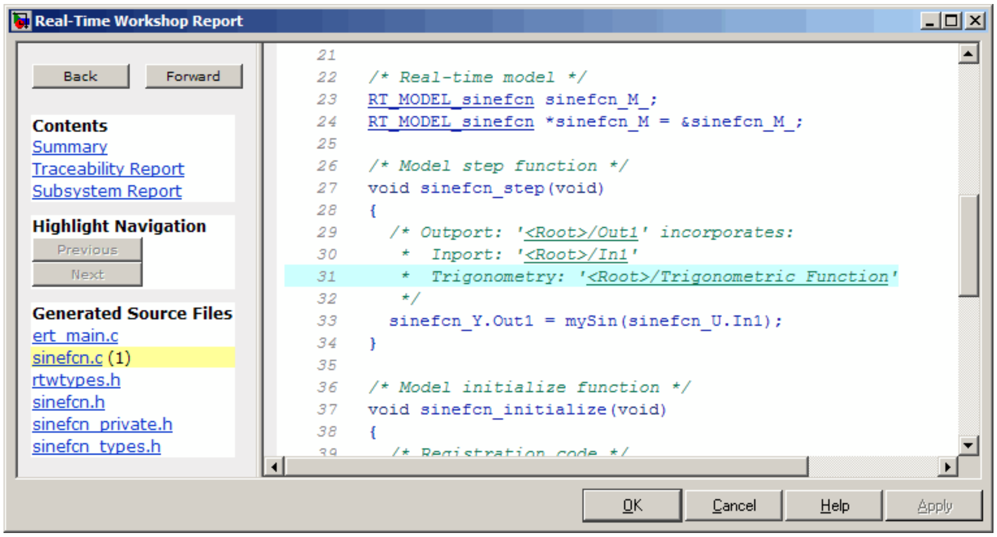
5 Create an ERT-based model with a Trigonometric Function block set to the sine function, such as the following:



Make sure that the TFL you registered, `Sine Function Example 2`, is selected for this model.

6 Go to the **Real-Time Workshop** pane of the Configuration Parameters dialog box and select the options **Generate HTML report** and **Block-to-code highlighting**. Select the **Generate code only** option and generate code for the model.

7 Go to the model window and use block-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the Trigonometric Function block and select **Real-Time Workshop > Highlight Code**. This highlights the `sin` function code within the model step function in `sinefcn.c`. In this case, `sin` has been replaced with `mySin` in the generated code.



```

/* Real-time model */
RT_MODEL_sinefcn sinefcn_M_;
RT_MODEL_sinefcn *sinefcn_M = &sinefcn_M_;

/* Model step function */
void sinefcn_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Trigonometry: '<Root>/Trigonometric Function'
   */
  sinefcn_Y.Out1 = mySin(sinefcn_U.In1);
}

/* Model initialize function */
void sinefcn_initialize(void)
{
  /* Registration code */
```

Example: Mapping Operators to Target-Specific Implementations

Real-Time Workshop Embedded Coder supports the following scalar operators for replacement with custom library functions using target function library (TFL) tables:

+ (addition)
- (subtraction)
* (multiplication)
/ (division)

This example uses the method described in "General Method for Creating Function and Operator Entries" on page 6-18 to create a TFL table entry for the + (addition) operator.

1 Create and save the following TFL table definition file, `tfl_table_add_uint8.m`. This file defines a TFL table containing an operator replacement entry for the + (addition) operator. The function body sets selected addition operator entry parameters, creates the `y1`, `u1`, and `u2` conceptual arguments individually, and then copies the conceptual arguments to the implementation arguments. Finally the operator entry is added to the table.

```
function hTable = tfl_table_add_unit8
%TFL_TABLE_ADD_UNIT8 - Describe operator entry for a Target Function Library table.

hTable = RTW.TflTable;

% Create entry for addition of built-in uint8 data type
% Saturation on, Rounding no preference
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                    'Key',                      'RTW_OP_ADD', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingMode',             'RTW_ROUND_UNSPECIFIED', ...
                    'ImplementationName',       'u8_add_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_add_u8_u8.c' );
```

```
arg = getTflArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);

arg = getTflArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(op_entry, arg );

arg = getTflArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(op_entry, arg );

copyConceptualArgsToImplementation(op_entry);

addEntry(hTable, op_entry);
```

2 Optionally, perform a quick check of the validity of the sine function entry by invoking the table definition file at the MATLAB command line (`>> tbl = tfl_table_add_uint8`) and by viewing it in the TFL Viewer (`>> RTW.viewTfl(tfl_table_add_uint8)`). For more information about validating TFL tables, see "Examining and Validating Function Replacement Tables" on page 6-58.

3 Create and save the following TFL registration file, which references the `tfl_table_add_uint8` table. The file specifies that the TFL to be registered is named `'Addition Operator Example'` and consists of `tfl_table_add_uint8`, with the default ANSI math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION


% Local function to define a TFL containing tfl_table_add_uint8
function thisTfl = locTflRegFcn

  % Instance a TFL registry entry
```

```
    thisTfl = RTW.TflRegistry;

    % Define the TFL properties
    thisTfl.Name = 'Addition Operator Example';
    thisTfl.Description = 'Demonstration of addition operator replacement';
    thisTfl.TableList = {'tfl_table_add_uint8'};
    thisTfl.BaseTfl = 'C89/C90 (ANSI)';
    thisTfl.TargetHWDeviceType = {'*'};

  end % End of LOCTFLREGFCN
```

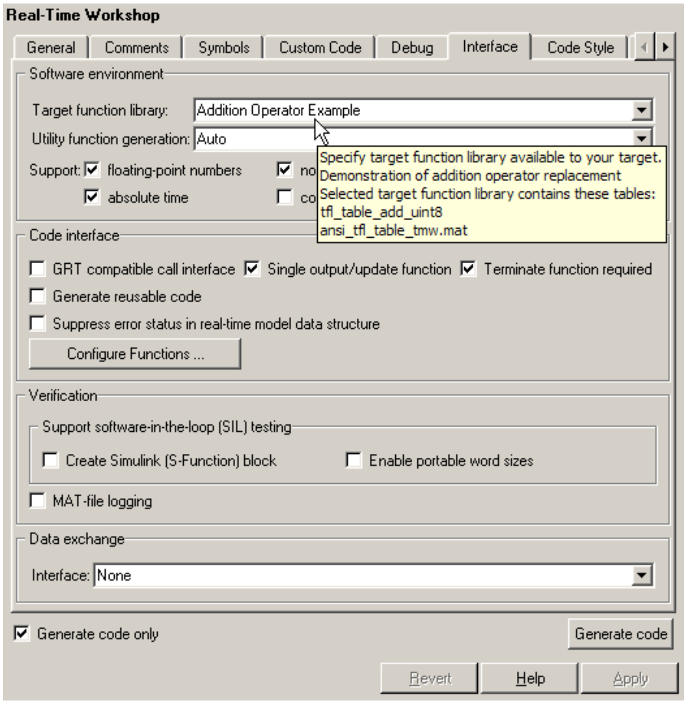
Place this `sl_customization.m` file in the MATLAB search path or in the current working directory, so that the TFL will be registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`.

For more information about using a TFL registration file, see "Registering Target Function Libraries" on page 6-66.

4 With your `sl_customization.m` file in the MATLAB search path or in the current working directory, open an ERT-based model in Simulink and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip is displayed containing information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Real-Time Workshop

General | Comments | Symbols | Custom Code | Debug | Interface | Code Style

Software environment

Target function library: Addition Operator Example

Utility function generation: Auto

Support: ☑ floating-point numbers ☑ no

☑ absolute time ☐ co

Specify target function library available to your target.
Demonstration of addition operator replacement
Selected target function library contains these tables:
tfl_table_add_uint8
ansi_tfl_table_tmw.mat

Code interface

☐ GRT compatible call interface ☑ Single output/update function ☑ Terminate function required

☐ Generate reusable code

☐ Suppress error status in real-time model data structure

Configure Functions ...

Verification

Support software-in-the-loop (SIL) testing

☐ Create Simulink (S-Function) block ☐ Enable portable word sizes

☐ MAT-file logging

Data exchange

Interface: None

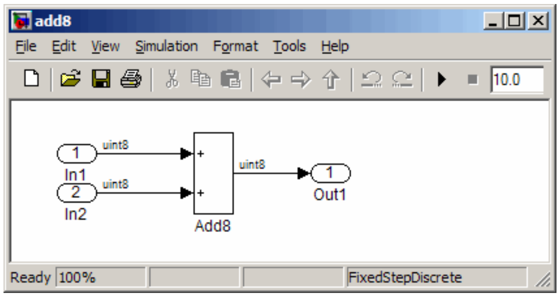
☑ Generate code only

Generate code

Revert Help Apply

Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Addition Operator Example`.

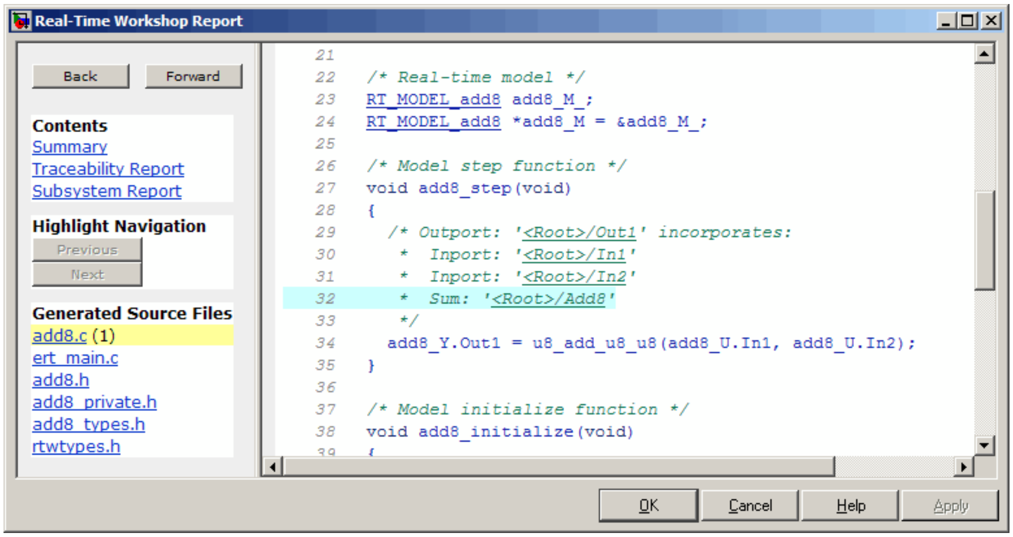
5 Create an ERT-based model with an Add block, such as the following:



Make sure that the TFL you registered, `Addition Operator Example`, is selected for this model.

6 Go to the **Real-Time Workshop** pane of the Configuration Parameters dialog box and select the options **Generate HTML report** and **Block-to-code highlighting**. Select the **Generate code only** option and generate code for the model.

7 Go to the model window and use block-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the Add block and select **Real-Time Workshop > Highlight Code**. This highlights the Sum block code within the model step function in `add8.c`. In this case, code containing the + operator has been replaced with `u8_add_u8_u8` in the generated code.



```

/* Real-time model */
RT_MODEL_add8 add8_M_;
RT_MODEL_add8 *add8_M = &add8_M_;

/* Model step function */
void add8_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Sum: '<Root>/Add8'
   */
  add8_Y.Out1 = u8_add_u8_u8(add8_U.In1, add8_U.In2);
}

/* Model initialize function */
void add8_initialize(void)
{
```

Mapping Fixed-Point Operators to Target-Specific Implementations

Overview

Real-Time Workshop Embedded Coder supports TFL-based function replacement for the following scalar operations on fixed-point data types:

+ (addition)
− (subtraction)
* (multiplication)
/ (division)

Fixed-point operator table entries can be defined as matching:

- A specific binary-point-only scaling combination on the operator inputs and output.
- A specific [slope bias] scaling combination on the operator inputs and output.
- Relative scaling between multiplication or division operator inputs and output.

  Use this method to map a range of slope and bias values to a replacement function for multiplication or division.

- Equal slope and zero net bias across addition or subtraction operator inputs and output.

  Use this method to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

Note

- The demo `rtwdemo_tflfixpt` demonstrates these replacements and provides example tables that can be used as a starting point for customization.
- Using fixed-point data types in a model requires a Simulink Fixed Point license.
- The fixed-point terminology used in this section is defined and explained in the *Simulink Fixed Point User's Guide*. See especially "Fixed-Point Numbers" and "Arithmetic Operations".

Fixed-Point Numbers and Arithmetic

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V = \tilde{V} = SQ + B$$

where

- V is an arbitrarily precise real-world value.
- $\tilde{V}$ is the approximate real-world value that results from fixed-point representation.
- Q is an integer that encodes $\tilde{V}$, referred to as the *quantized integer*.
- S is a coefficient of Q, referred to as the *slope*.
- B is an additive correction, referred to as the *bias*.

The general equation for an operation between fixed-point operands is as follows:

$$(S_O Q_O + B_O) = (S_1 Q_1 + B_1) < op > (S_2 Q_2 + B_2)$$

The objective of TFL fixed-point operator replacement is to replace an operator that accepts and returns fixed-point or integer inputs and output with a function that accepts and returns built-in C numeric data types (not fixed-point data types). The following sections provide additional programming information for each supported operator.

Addition. The operation *V0 = V1 + V2* implies that

$$Q_0 = \left(\frac{S_1}{S_0}\right) Q_1 + \left(\frac{S_2}{S_0}\right) Q_2 + \left(\frac{B_1 + B_2 - B_0}{S_0}\right)$$

If an addition replacement function is defined such that the scaling on the operands and sum are equal and the net bias

$$\left(\frac{B_1 + B_2 - B_0}{S_0}\right)$$

is zero (for example, a function `s8_add_s8_s8` that adds two signed 8-bit values and produces a signed 8-bit result), then the TFL operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Subtraction. The operation *V0 = V1 − V2* implies that

$$Q_0 = \left(\frac{S_1}{S_0}\right) Q_1 - \left(\frac{S_2}{S_0}\right) Q_2 + \left(\frac{B_1 - B_2 - B_0}{S_0}\right)$$

If a subtraction replacement function is defined such that the scaling on the operands and difference are equal and the net bias

$$\left(\frac{B_1 - B_2 - B_0}{S_0}\right)$$

is zero (for example, a function `s8_sub_s8_s8` that subtracts two signed 8-bit values and produces a signed 8-bit result), then the TFL operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Multiplication. The operation *V0 = V1 * V2* implies, for binary-point-only scaling, that

$$S_0 Q_0 = (S_1 Q_1)(S_2 Q_2)$$
$$Q_0 = \left(\frac{S_1 S_2}{S_0}\right) Q_1 Q_1$$
$$Q_0 = S_n Q_1 Q$$

where S_n is the net slope.

Multiplication replacement functions may be defined such that all scaling is contained by a single operand. For example, a replacement function `s8_mul_s8_u8_rsf0p125` can multiply a signed 8-bit value by a factor of `[0 ... 0.1245]` and produce a signed 8-bit result. The following discussion describes how to convert the slope on each operand into a net factor.

To match a multiplication operation to the `s8_mul_s8_u8_rsf0p125` replacement function, $0 <= S_n Q_2 <= 2^{-3}$. Substituting the maximum integer value for Q_2 results in the following match criteria: When $S_n 2^8 = 2^{-3}$, or $S_n = 2^{-11}$, TFL replacement processing will map the multiplication operation to the `s8_mul_s8_u8_rsf0p125` function.

To accomplish this mapping, the TFL operator entry must define a *relative scaling factor*, $F2^E$, where the values for *F* and *E* are provided using operator entry parameters `RelativeScalingFactorF` and `RelativeScalingFactorE`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.) For the `s8_mul_s8_u8_rsf0p125` function, the `RelativeScalingFactorF` would be set to 1 and the `RelativeScalingFactorE` would be set to -3.

Note When an operator entry specifies `RelativeScalingFactorF` and `RelativeScalingFactorE`, zero bias is implied for the inputs and output.

Division. The operation *V0* = *(V1 / V2)* implies, for binary-point-only scaling, that

$$S_0Q_0 = \left(\frac{S_1Q_1}{S_2Q_2}\right)$$

$$Q_0 = S_n\left(\frac{Q_1}{Q_2}\right)$$

where S_n is the net slope.

As with multiplication, division replacement functions may be defined such that all scaling is contained by a single operand. For example, a replacement function `s16_rsf0p5_div_s16_s16` can divide a signed 16<<16 value by a signed 16-bit value and produce a signed 16-bit result. The following discussion describes how to convert the slope on each operand into a net factor.

To match a division operation to the `s16_rsf0p5_div_s16_s16` replacement function, $0 <= S_nQ_1 <= 2^{-1}$. Substituting the maximum integer value for Q_1 results in the following match criteria: When $S_n2^{15} = 2^{-1}$, or $S_n = 2^{-16}$, TFL replacement processing will map the division operation to the `s8_mul_s8_u8_rsf0p125` function.

To accomplish this mapping, the TFL operator entry must define a *relative scaling factor*, $F2^E$, where the values for *F* and *E* are provided using operator entry parameters `RelativeScalingFactorF` and `RelativeScalingFactorE`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.) For the `s16_rsf0p5_div_s16_s16` function, the `RelativeScalingFactorF` would be set to 1 and the `RelativeScalingFactorE` would be set to -1.

Note When an operator entry specifies `RelativeScalingFactorF` and `RelativeScalingFactorE`, zero bias is implied for the inputs and output.

Creating Fixed-Point Operator Entries

To create TFL table entries for fixed-point operators, you use the "General Method for Creating Function and Operator Entries" on page 6-18 and specify fixed-point parameter/value pairs to the following functions:

Function	Description
`createAndAddConceptualArg`	Create conceptual argument from specified properties and add to conceptual arguments for TFL table entry
`createAndAddImplementationArg`	Create implementation argument from specified properties and add to implementation arguments for TFL table entry
`createAndSetCImplementationReturn`	Create implementation return argument from specified properties and add to implementation for TFL table entry
`setTflCOperationEntryParameters`	Set specified parameters for operator entry in TFL table

The following table maps some common methods of matching TFL fixed-point operator table entries to the associated fixed-point parameters that you need to specify in your TFL table definition file:

To match...	Instantiate class...	Minimally specify parameters...
A specific binary-point-only scaling combination on the operator inputs and output (See "Example: Creating Fixed-Point Operator Entries for Binary-Point-Only Scaling" on page 6-41)	`RTW.TflCOperationEntry`	`createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `true`. • `CheckBias`: Specify the value `true`. • `DataTypeMode` (or `DataType/Scaling` equivalent): Specify fixed-point binary-point-only scaling. • `FractionLength`: Specify a fraction length (for example, 3).
A specific [slope bias] scaling combination on the operator inputs and output (See "Example: Creating Fixed-Point Operator Entries for [Slope Bias] Scaling" on page 6-44)	`RTW.TflCOperationEntry`	`createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `true`. • `CheckBias`: Specify the value `true`. • `DataTypeMode` (or `DataType/Scaling` equivalent): Specify fixed-point [slope bias] scaling. • `Slope` (or `SlopeAdjustmentFactor/-FixedExponent` equvalent): Specify a slope value (for example, 15). • `Bias`: Specify a bias value (for example, 2).

To match...	Instantiate class...	Minimally specify parameters...
Relative scaling between operator inputs and output (multiplication and division) (See “Example: Creating Fixed-Point Operator Entries for Relative Scaling (Multiplication and Division)” on page 6-47)	`RTW.TflCOperationEntry-Generator`	`setTflCOperationEntryParameters` function: • `RelativeScalingFactorF`: Specify the slope adjustment factor (`F`) part of the relative scaling factor, $\texttt{F2}^{\texttt{E}}$ (for example, `1.0`). • `RelativeScalingFactorE`: Specify the fixed exponent (`E`) part of the relative scaling factor, $\texttt{F2}^{\texttt{E}}$ (for example, `-3.0`). `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`. • `DataType`: Specify the value `'Fixed'`.
Equal slope and zero net bias across operator inputs and output (addition and subtraction) (See “Example: Creating Fixed-Point Operator Entries for Equal Slope and Zero Net Bias (Addition and Subtraction)” on page 6-50)	`RTW.TflCOperationEntry-Generator`	`setTflCOperationEntryParameters` function: • `SlopesMustBeTheSame`: Specify the value `true`. • `MustHaveZeroNetBias`: Specify the value `true`. `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`.

Example: Creating Fixed-Point Operator Entries for Binary-Point-Only Scaling

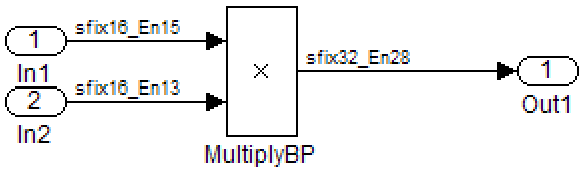
TFL table entries for operations on fixed-point data types can be defined as matching a specific binary-point-only scaling combination on the operator inputs and output. These binary-point-only scaling entries can map the specified binary-point-scaling combination to a replacement function for addition, subtraction, multiplication, or division.

This example uses the method described in "General Method for Creating Function and Operator Entries" on page 6-18 to create a TFL table entry for multiplication of fixed-point data types where arguments are specified with binary-point-only scaling. In this example:

- The TFL operator entry is instantiated using the `RTW.TflCOperationEntry` class.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters, including the type of operation (multiplication), the saturation mode (saturate on overflow), the rounding mode (unspecified), and the name of the replacement function (`s32_mul_s16_s16_binarypoint`).
- The function `createAndAddConceptualArg` is called to create and add conceptual output and input arguments to the operator entry. Each argument specifies that the data type is fixed-point, the mode is binary-point-only scaling, and its derived slope and bias values must exactly match the call-site slope and bias values. The output argument is 32 bits, signed, with a fraction length of 28, while the input arguments are 16 bits, signed, with fraction lengths of 15 and 13.
- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output argument is 32 bits and signed (`int32`) and the input arguments are 16 bits and signed (`int16`).

```
hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                    'Key',                       'RTW_OP_MUL', ...
                    'Priority',                  90, ...
                    'SaturationMode',            'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingMode',              'RTW_ROUND_UNSPECIFIED', ...
                    'ImplementationName',        's32_mul_s16_s16_binarypoint', ...
                    'ImplementationHeaderFile', 's32_mul_s16_s16_binarypoint.h', ...
                    'ImplementationSourceFile', 's32_mul_s16_s16_binarypoint.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric',...
                          'Name',           'y1', ...
                          'IOType',         'RTW_IO_OUTPUT', ...
                          'CheckSlope',     true, ...
                          'CheckBias',      true, ...
                          'DataTypeMode',   'Fixed-point: binary point scaling', ...
                          'IsSigned',       true, ...
                          'WordLength',     32, ...
                          'FractionLength', 28);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'u1', ...
                          'IOType',         'RTW_IO_INPUT', ...
                          'CheckSlope',     true, ...
                          'CheckBias',      true, ...
                          'DataTypeMode',   'Fixed-point: binary point scaling', ...
                          'IsSigned',       true, ...
                          'WordLength',     16, ...
                          'FractionLength', 15);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'u2', ...
                          'IOType',         'RTW_IO_INPUT', ...
                          'CheckSlope',     true, ...
                          'CheckBias',      true, ...
                          'DataTypeMode',   'Fixed-point: binary point scaling', ...
                          'IsSigned',       true, ...
                          'WordLength',     16, ...
                          'FractionLength', 13);

createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                              'Name',           'y1', ...
                              'IOType',         'RTW_IO_OUTPUT', ...
                              'IsSigned',       true, ...
                              'WordLength',     32, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',           'u1', ...
                              'IOType',         'RTW_IO_INPUT', ...
                              'IsSigned',       true, ...
                              'WordLength',     16, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',           'u2', ...
                              'IOType',         'RTW_IO_INPUT', ...
                              'IsSigned',       true, ...
                              'WordLength',     16, ...
                              'FractionLength', 0);

addEntry(hTable, op_entry);
```

To generate code using this table entry, you can follow the general procedure in “Example: Mapping Operators to Target-Specific Implementations” on page 6-28, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to `fixdt(1,16,15)`
- Set the Inport 2 **Data type** to `fixdt(1,16,13)`
- Set the Product block **Output data type** to `fixdt(1,32,28)` and select the option **Saturate on integer overflow**

Example: Creating Fixed-Point Operator Entries for [Slope Bias] Scaling

TFL table entries for operations on fixed-point data types can be defined as matching a specific [slope bias] scaling combination on the operator inputs and output. These [slope bias] scaling entries can map the specified [slope bias] combination to a replacement function for addition, subtraction, multiplication, or division.

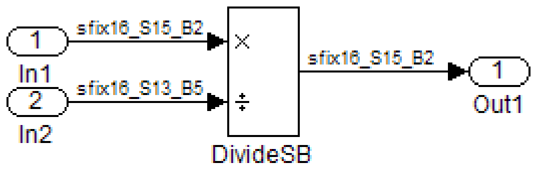
This example uses the method described in "General Method for Creating Function and Operator Entries" on page 6-18 to create a TFL table entry for division of fixed-point data types where arguments are specified using [slope bias] scaling. In this example:

- The TFL operator entry is instantiated using the `RTW.TflCOperationEntry` class.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters, including the type of operation (division), the saturation mode (saturate on overflow), the rounding mode (round to ceiling), and the name of the replacement function (`s16_div_s16_s16_slopebias`).
- The function `createAndAddConceptualArg` is called to create and add conceptual output and input arguments to the operator entry. Each argument specifies that the data type is fixed-point, the mode is [slope bias] scaling, and its specified slope and bias values must exactly match the call-site slope and bias values. The output argument and input arguments are 16 bits, signed, each with specific [slope bias] specifications.
- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```
hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                    'Key',                     'RTW_OP_DIV', ...
                    'Priority',                90, ...
```

```
                        'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                        'RoundingMode',             'RTW_ROUND_CEILING', ...
                        'ImplementationName',       's16_div_s16_s16_slopebias', ...
                        'ImplementationHeaderFile', 's16_div_s16_s16_slopebias.h', ...
                        'ImplementationSourceFile', 's16_div_s16_s16_slopebias.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'y1', ...
                          'IOType',         'RTW_IO_OUTPUT', ...
                          'CheckSlope',     true, ...
                          'CheckBias',      true, ...
                          'DataTypeMode',   'Fixed-point: slope and bias scaling', ...
                          'IsSigned',       true, ...
                          'WordLength',     16, ...
                          'Slope',          15, ...
                          'Bias',           2);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'u1', ...
                          'IOType',         'RTW_IO_INPUT', ...
                          'CheckSlope',     true, ...
                          'CheckBias',      true, ...
                          'DataTypeMode',   'Fixed-point: slope and bias scaling', ...
                          'IsSigned',       true, ...
                          'WordLength',     16, ...
                          'Slope',          15, ...
                          'Bias',           2);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'u2', ...
                          'IOType',         'RTW_IO_INPUT', ...
                          'CheckSlope',     true, ...
                          'CheckBias',      true, ...
                          'DataTypeMode',   'Fixed-point: slope and bias scaling', ...
                          'IsSigned',       true, ...
                          'WordLength',     16, ...
                          'Slope',          13, ...
                          'Bias',           5);

createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                              'Name',          'y1', ...
                              'IOType',        'RTW_IO_OUTPUT', ...
                              'IsSigned',      true, ...
                              'WordLength',    16, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',          'u1', ...
                              'IOType',        'RTW_IO_INPUT', ...
                              'IsSigned',      true, ...
                              'WordLength',    16, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',          'u2', ...
                              'IOType',        'RTW_IO_INPUT', ...
                              'IsSigned',      true, ...
                              'WordLength',    16, ...
                              'FractionLength', 0);

addEntry(hTable, op_entry);
```

To generate code using this table entry, you can follow the general procedure in “Example: Mapping Operators to Target-Specific Implementations” on page 6-28, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to `fixdt(1,16,15,2)`
- Set the Inport 2 **Data type** to `fixdt(1,16,13,5)`

- Set the Divide block **Output data type** to `Inherit:  Inherit via back propagation`, set **Round integer calculations toward** to `Ceiling`, and select the option **Saturate on integer overflow**

Example: Creating Fixed-Point Operator Entries for Relative Scaling (Multiplication and Division)

TFL table entries for multiplication or division of fixed-point data types can be defined as matching relative scaling between operator inputs and output. These relative scaling entries can map a range of slope and bias values to a replacement function for multiplication or division.

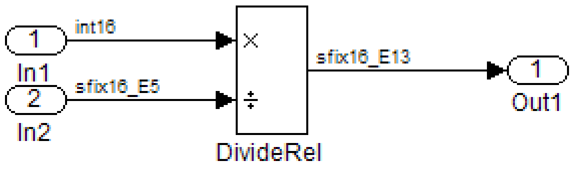
This example uses the method described in "General Method for Creating Function and Operator Entries" on page 6-18 to create a TFL table entry for division of fixed-point data types using a relative scaling factor. In this example:

- The TFL operator entry is instantiated using the `RTW.TflCOperationEntryGenerator` class, which provides access to the fixed-point parameters `RelativeScalingFactorF` and `RelativeScalingFactorE`.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters, including the type of operation (division), the saturation mode (saturation off), the rounding mode (round to ceiling), and the name of the replacement function (`s16_div_s16_s16_rsf0p125`). Additionally, `RelativeScalingFactorF` and `RelativeScalingFactorE` are used to specify the `F` and `E` parts of the relative scaling factor $F2^E$.
- The function `createAndAddConceptualArg` is called to create and add conceptual output and input arguments to the operator entry. Each argument is specified as fixed-point, 16 bits, and signed. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.
- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```
hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntryGenerator;
setTflCOperationEntryParameters(op_entry, ...
                    'Key',                        'RTW_OP_DIV', ...
                    'Priority',                   90, ...
                    'SaturationMode',             'RTW_WRAP_ON_OVERFLOW', ...
                    'RoundingMode',               'RTW_ROUND_CEILING', ...
                    'RelativeScalingFactorF',     1.0, ...
                    'RelativeScalingFactorE',     -3.0, ...
                    'ImplementationName',         's16_div_s16_s16_rsf0p125', ...
                    'ImplementationHeaderFile', 's16_div_s16_s16_rsf0p125.h', ...
                    'ImplementationSourceFile', 's16_div_s16_s16_rsf0p125.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'y1', ...
                          'IOType',         'RTW_IO_OUTPUT', ...
                          'CheckSlope',     false, ...
                          'CheckBias',      false, ...
                          'DataType',       'Fixed', ...
                          'IsSigned',       true, ...
                          'WordLength',     16);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'u1', ...
                          'IOType',         'RTW_IO_INPUT', ...
                          'CheckSlope',     false, ...
                          'CheckBias',      false, ...
                          'DataType',       'Fixed', ...
                          'IsSigned',       true, ...
                          'WordLength',     16);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'u2', ...
                          'IOType',         'RTW_IO_INPUT', ...
                          'CheckSlope',     false, ...
                          'CheckBias',      false, ...
                          'DataType',       'Fixed', ...
                          'IsSigned',       true, ...
                          'WordLength',     16);
```

```
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                              'Name',           'y1', ...
                              'IOType',         'RTW_IO_OUTPUT', ...
                              'IsSigned',       true, ...
                              'WordLength',     16, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',           'u1', ...
                              'IOType',         'RTW_IO_INPUT', ...
                              'IsSigned',       true, ...
                              'WordLength',     16, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',           'u2', ...
                              'IOType',         'RTW_IO_INPUT', ...
                              'IsSigned',       true, ...
                              'WordLength',     16, ...
                              'FractionLength', 0);

addEntry(hTable, op_entry);
```

To generate code using this table entry, you can follow the general procedure in “Example: Mapping Operators to Target-Specific Implementations” on page 6-28, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to `int16`

- Set the Inport 2 **Data type** to `fixdt(1,16,-5)`
- Set the Divide block **Output data type** to `fixdt(1,16,-13)` and set **Round integer calculations toward** to `Ceiling`

Example: Creating Fixed-Point Operator Entries for Equal Slope and Zero Net Bias (Addition and Subtraction)

TFL table entries for addition or subtraction of fixed-point data types can be defined as matching relative slope and bias values (equal slope and zero net bias) across operator inputs and output. These entries allow you to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

This example uses the method described in "General Method for Creating Function and Operator Entries" on page 6-18 to create a TFL table entry for addition of fixed-point data types where slopes must be equal and net bias must be zero across the operator inputs and output. In this example:

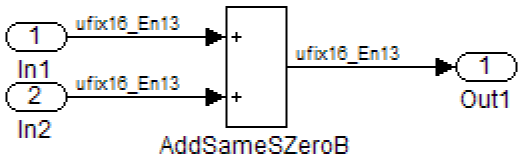
- The TFL operator entry is instantiated using the `RTW.TflCOperationEntryGenerator` class, which provides access to the fixed-point parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias`.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters, including the type of operation (addition), the saturation mode (saturation off), the rounding mode (unspecified), and the name of the replacement function (`u16_add_SameSlopeZeroBias`). Additionally, `SlopesMustBeTheSame` and `MustHaveZeroNetBias` are set to `true` to indicate that slopes must be equal and net bias must be zero across the addition inputs and output.
- The function `createAndAddConceptualArg` is called to create and add conceptual output and input arguments to the operator entry. Each argument is specified as 16 bits and unsigned. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.
- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types

(not fixed-point data types). In this case, the output and input arguments are 16 bits and unsigned (`uint16`).

```
hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntryGenerator;
setTflCOperationEntryParameters(op_entry, ...
                    'Key',                      'RTW_OP_ADD', ...
                    'Priority',                  90, ...
                    'SaturationMode',            'RTW_WRAP_ON_OVERFLOW', ...
                    'RoundingMode',              'RTW_ROUND_UNSPECIFIED', ...
                    'SlopesMustBeTheSame',       true, ...
                    'MustHaveZeroNetBias',       true, ...
                    'ImplementationName',        'u16_add_SameSlopeZeroBias', ...
                    'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
                    'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'y1', ...
                          'IOType',         'RTW_IO_OUTPUT', ...
                          'CheckSlope',     false, ...
                          'CheckBias',      false, ...
                          'IsSigned',       false, ...
                          'WordLength',     16);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'u1', ...
                          'IOType',         'RTW_IO_INPUT', ...
                          'CheckSlope',     false, ...
                          'CheckBias',      false, ...
                          'IsSigned',       false, ...
                          'WordLength',     16);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'u2', ...
                          'IOType',         'RTW_IO_INPUT', ...
                          'CheckSlope',     false, ...
                          'CheckBias',      false, ...
                          'IsSigned',       false, ...
                          'WordLength',     16);
```

```
   createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                              'Name',          'y1', ...
                              'IOType',        'RTW_IO_OUTPUT', ...
                              'IsSigned',      false, ...
                              'WordLength',    16, ...
                              'FractionLength', 0);

   createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',          'u1', ...
                              'IOType',        'RTW_IO_INPUT', ...
                              'IsSigned',      false, ...
                              'WordLength',    16, ...
                              'FractionLength', 0);

   createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',          'u2', ...
                              'IOType',        'RTW_IO_INPUT', ...
                              'IsSigned',      false, ...
                              'WordLength',    16, ...
                              'FractionLength', 0);

   addEntry(hTable, op_entry);
```

To generate code using this table entry, you can follow the general procedure in "Example: Mapping Operators to Target-Specific Implementations" on page 6-28, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to `fixdt(0,16,13)`
- Set the Inport 2 **Data type** to `fixdt(0,16,13)`

- Verify that the Add block **Output data type** is set to its default, `Inherit via internal rule`, and set **Round integer calculations toward** to `Zero`

Specifying Build Information for Function Replacements

- "Functions for Specifying Table Entry Build Information" on page 6-53
- "Using RTW.copyFileToBuildDir to Copy Files to the Build Directory" on page 6-54
- "RTW.copyFileToBuildDir Examples" on page 6-55

Functions for Specifying Table Entry Build Information

As you create TFL table entries for function or operator replacement, you specify the header and source file information for each function implementation using one of the following:

- The arguments `ImplementationHeaderFile`, `ImplementationHeaderPath`, `ImplementationSourceFile`, and `ImplementationSourcePath` to `setTflCFunctionEntryParameters` or `setTflCOperationEntryParameters`
- The `headerFile` argument to `registerCFunctionEntry` or `registerCPromotableMacroEntry`

Also, each table entry can specify additional header files, source files, and object files to be included in model builds whenever the TFL table entry is matched and used to replace a function or operator in generated code. To add an additional header file, source file, or object file, use the following TFL table creation functions:

Function	Description
`addAdditionalHeaderFile`	Add additional header file to array of additional header files for TFL table entry
`addAdditionalIncludePath`	Add additional include path to array of additional include paths for TFL table entry

Function	Description
`addAdditionalLinkObj`	Add additional link object to array of additional link objects for TFL table entry
`addAdditionalLinkObjPath`	Add additional link object path to array of additional link object paths for TFL table entry
`addAdditionalSourceFile`	Add additional source file to array of additional source files for TFL table entry
`addAdditionalSourcePath`	Add additional source path to array of additional source paths for TFL table entry

For function descriptions and examples, see the function reference pages in the Real-Time Workshop Embedded Coder reference documentation.

Using RTW.copyFileToBuildDir to Copy Files to the Build Directory

If a TFL table entry uses header, source, or object files that reside in external directories, and if the table entry is matched and used to replace a function or operator in generated code, the external files will need to be copied to the build directory before the generated code can be built. The `RTW.copyFileToBuildDir` function can be invoked after code generation to copy the table entry's specified header file, source file, additional header files, additional source files, and additional link objects to the build directory. The copied files will be available for use in the build process.

To direct that a table entry's external files should be copied to the build directory after code generation, specify the argument `'RTW.copyFileToBuildDir'` to the `genCallback` parameter of the TFL function that you use to set the table entry parameters, among the following:

- `registerCFunctionEntry`
- `registerCPromotableMacroEntry`
- `setTflCFunctionEntryParameters`
- `setTflCOperationEntryParameters`

RTW.copyFileToBuildDir Examples

The following example defines a table entry for an optimized multiplication function that takes signed 32-bit integers and returns a signed 32-bit integer, taking saturation into account. Multiplications in the generated code are to be replaced with calls to your optimized function. Your optimized function resides in an external directory and must be copied into the build directory to be compiled and linked into the application.

The multiplication table entry specifies the source and header file names as well as their full paths. To request the copy to be performed, the table entry specifies the argument `'RTW.copyFileToBuildDir'` to the `genCallback` parameter of the `setTflCOperationEntryParameters` function. In this example, the header file `s32_mul.h` contains an inlined function that invokes assembly functions contained in `s32_mul.s`. If the table entry is matched and used to generate code, the `RTW.copyFileToBuildDir` function will copy the specified source and header files into the build directory.

```
function hTable = make_my_tfl_table

hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                'Key',                      'RTW_OP_MUL', ...
                'Priority',                 100, ...
                'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                'RoundingMode',             'RTW_ROUND_UNSPECIFIED', ...
                'ImplementationName',       's32_mul_s32_s32_sat', ...
                'ImplementationHeaderFile', 's32_mul.h', ...
                'ImplementationSourceFile', 's32_mul.s', ...
                'ImplementationHeaderPath', {fullfile('$(MATLAB_ROOT)','tfl')}, ...
                'ImplementationSourcePath', {fullfile('$(MATLAB_ROOT)','tfl')}, ...
                'GenCallback',              'RTW.copyFileToBuildDir');
.
.
.
addEntry(hTable, op_entry);
```

The following example shows the use of the `addAdditional*` functions along with `RTW.copyFileToBuildDir`.

```
hTable = RTW.TflTable;

% Path to external source, header, and object files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                'Key',                        'RTW_OP_ADD', ...
                'Priority',                   90, ...
                'SaturationMode',             'RTW_SATURATE_UNSPECIFIED', ...
                'RoundingMode',               'RTW_ROUND_UNSPECIFIED', ...
                'ImplementationName',         's32_add_s32_s32', ...
                'ImplementationHeaderFile', 's32_add_s32_s32.h', ...
                'ImplementationSourceFile', 's32_add_s32_s32.c'...
                'GenCallback',                'RTW.copyFileToBuildDir');

addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));
addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjsPath(op_entry, fullfile(libdir, 'bin'));
.
.
.
addEntry(hTable, op_entry);
```

Adding TFL Reserved Identifiers

Real-Time Workshop reserves certain words for its own use as keywords of the generated code language. Real-Time Workshop keywords are reserved for use internal to Real-Time Workshop or C programming, and should not be used in Simulink models as identifiers or function names. Real-Time Workshop reserved keywords include many TFL identifiers, the majority of which are function names, such as `acos`. To view the base list of TFL reserved identifiers, see "Reserved Keywords" in the Real-Time Workshop documentation.

In a TFL table, each function implementation name defined by a table entry will be registered as a reserved identifier. You can register additional reserved identifiers for the table on a per-header-file basis. Providing additional reserved identifiers can help prevent duplicate symbols and other identifier-related compile and link issues.

To register additional TFL reserved identifiers, use the following function:

Function	Description
`setReservedIdentifiers`	Register specified reserved identifiers to be associated with TFL table

You can register up to four reserved identifier structures in a TFL table. One set of reserved identifiers can be associated with an arbitrary TFL, while the other three (if present) must be associated with ANSI, ISO, or GNU libraries. Here is an example of a reserved identifier structure that specifies two identifiers and the associated header file.

```
d{1}.LibraryName = 'ANSI';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

The specified identifiers are added to the reserved identifiers collection and honored during the Real-Time Workshop build procedure. For more information and examples, see `setReservedIdentifiers` in the Real-Time Workshop Embedded Coder reference documentation.

Examining and Validating Function Replacement Tables

In this section...

Overview

After you create a target function library (TFL) table containing your function replacement entries, and before you deploy production TFLs containing your table for general use in building models, you can use various techniques to examine and validate the TFL table entries. These include

- Invoking the table definition M-file
- Using the TFL Viewer at various stages of TFL development to examine TFLs, tables, and entries
- Tracing code generated from models for which your TFL is selected
- Examining TFL cache hits and misses logged during code generation

Invoking the Table Definition M-File

Immediately after creating or modifying a table definition M-file (as described in "Creating Function Replacement Tables" on page 6-13), you should invoke it at the MATLAB command line. This serves as a check of the validity of your table entries. For example,

```
>> tbl = tfl_table_sinfcn

tbl =

RTW.TflTable
            Version: '1.0'
```

```
         AllEntries: [2x1 RTW.TflCFunctionEntry]
    ReservedSymbols: []

>>
```

Any errors found during the invocation will be displayed. In the following example, a typo in a data type name is detected and displayed.

```
>> tbl = tfl_table_sinfcn
??? RTW_CORE:tfl:TflTable: Unsupported data type, 'dooble'.

Error in ==> tfl_table_sinfcn at 7
hTable.registerCFunctionEntry(100, 1, 'sin', 'dooble', 'sin_dbl', ...

>>
```

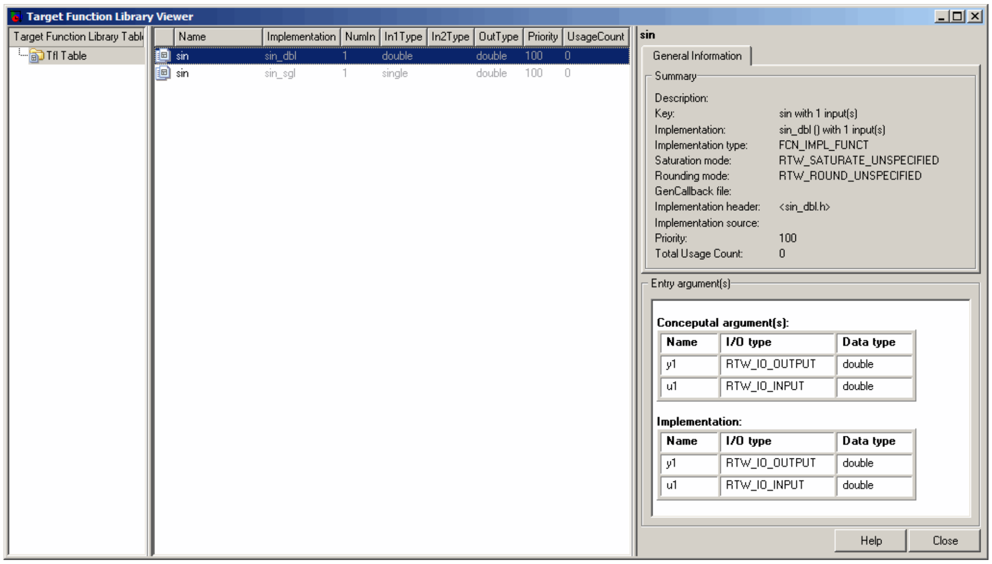
Using the TFL Viewer to Examine Your Table

After creating or modifying a table definition M-file, as a further check of your table entries, you should use the TFL Viewer to display and examine your table. Invoke the TFL Viewer using the following form of the MATLAB command `RTW.viewTfl`:

```
RTW.viewTfl(table-name)
```

For example,

```
>> RTW.viewTfl(tfl_table_sinfcn)
```



Name	Implementation	NumIn	In1Type	In2Type	OutType	Priority	UsageCount
sin	sin_dbl	1	double		double	100	0
sin	sin_sgl	1	single		double	100	0

sin

Summary

Description:	
Key:	sin with 1 input(s)
Implementation:	sin_dbl () with 1 input(s)
Implementation type:	FCN_IMPL_FUNCT
Saturation mode:	RTW_SATURATE_UNSPECIFIED
Rounding mode:	RTW_ROUND_UNSPECIFIED
GenCallback file:	
Implementation header:	<sin_dbl.h>
Implementation source:	
Priority:	100
Total Usage Count:	0

Entry argument(s)

Conceptual argument(s):

Name	I/O type	Data type
y1	RTW_IO_OUTPUT	double
u1	RTW_IO_INPUT	double

Implementation:

Name	I/O type	Data type
y1	RTW_IO_OUTPUT	double
u1	RTW_IO_INPUT	double

Select entries in your table and verify that the graphical display of the contents of your table meets your expectations. Common problems that can be detected at this stage include

- Incorrect argument order
- Conceptual argument naming that does not match the naming convention used by the code generation process
- Incorrect relative priority of entries within the table (highest priority is 0, and lowest priority is 100).

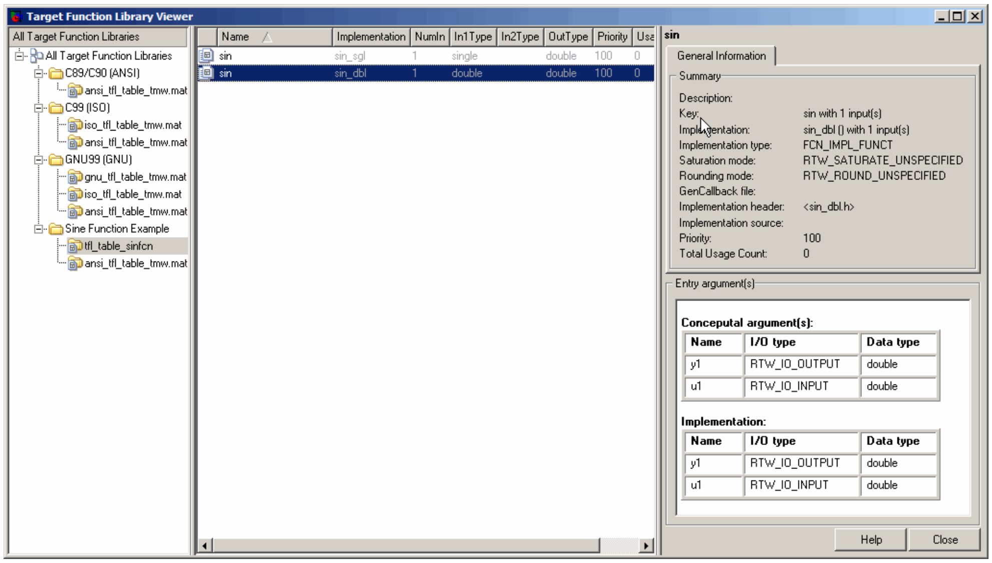
For more information about the TFL Viewer, see “Using the Target Function Library Viewer” in the Real-Time Workshop documentation.

Using the TFL Viewer to Examine Registered TFLs

After you register a TFL that includes your function replacement table (as described in “Registering Target Function Libraries” on page 6-66), you should use the TFL Viewer to verify that your TFL was properly registered and to examine the TFL and the tables it contains. Invoke the TFL Viewer using the

MATLAB command `RTW.viewTfl` with no arguments. This will display all TFLs registered in the current Simulink session. For example,

```
>> RTW.viewTfl
```



Name	Implementation	NumIn	In1Type	In2Type	OutType	Priority	Usa
sin	sin_sgl	1	single		double	100	0
sin	sin_dbl	1	double		double	100	0

Summary

Description:	
Key:	sin with 1 input(s)
Implementation:	sin_dbl () with 1 input(s)
Implementation type:	FCN_IMPL_FUNCT
Saturation mode:	RTW_SATURATE_UNSPECIFIED
Rounding mode:	RTW_ROUND_UNSPECIFIED
GenCallback file:	
Implementation header:	<sin_dbl.h>
Implementation source:	
Priority:	100
Total Usage Count:	0

Entry argument(s)

Conceptual argument(s):

Name	I/O type	Data type
y1	RTW_IO_OUTPUT	double
u1	RTW_IO_INPUT	double

Implementation:

Name	I/O type	Data type
y1	RTW_IO_OUTPUT	double
u1	RTW_IO_INPUT	double

If your TFL is not displayed, there may be an error in your `sl_customization.m` file, or you may need to issue the MATLAB command `sl_refresh_customizations`.

If your TFL is displayed, select the TFL and examine and compare its tables, including their relative order. Common problems that can be detected at this stage include

- Incorrect relative order of tables in the library (tables are displayed in search order)
- Table entry problems as listed in the previous section

For more information about the TFL Viewer, see "Using the Target Function Library Viewer" in the Real-Time Workshop documentation.

)

Tracing Code Generated Using Your TFL

After you register a TFL that includes your function replacement tables, you should use the TFL to generate code and verify that you are obtaining the function or operator replacement that you expect. For example, the following approach uses block-to-code highlighting to trace a specific expected replacement.

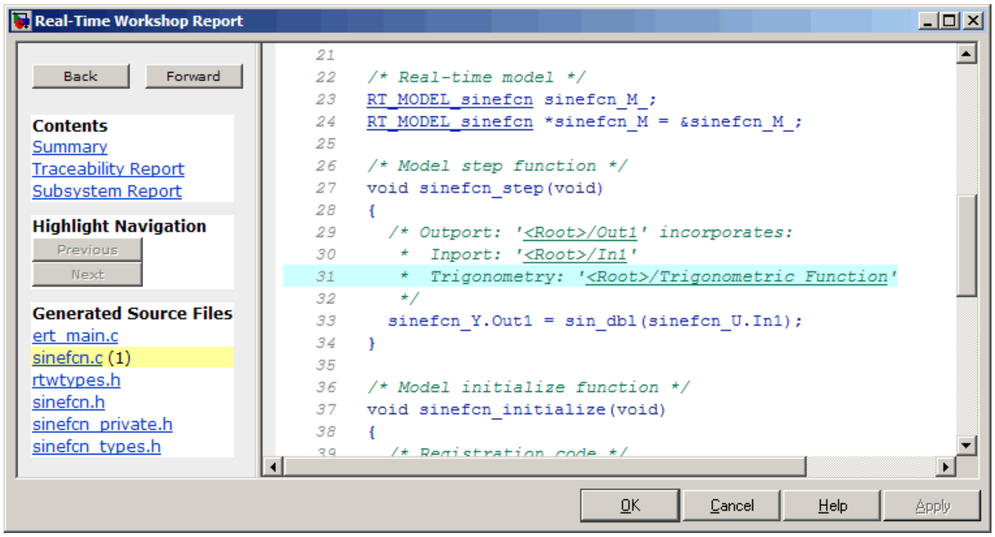
1 Open a ERT-based model for which you believe function or operator replacement should occur.

2 Select your TFL in the **Target function library** drop-down list on the **Interface** pane of the Configuration Parameters dialog box.

3 Select the options **Generate HTML report** and **Block-to-code highlighting** on the **Real-Time Workshop** pane of the Configuration Parameters dialog box.

4 Generate code for your model.

5 Go to the model window and use block-to-code highlighting to trace the code generated using your TFL. For example, right-click a block that you expect to have generated a function or operator replacement and select **Real-Time Workshop > Highlight Code**. This highlights the applicable generated function code within the HTML report. For example,



```
/* Real-time model */
RT_MODEL_sinefcn sinefcn_M_;
RT_MODEL_sinefcn *sinefcn_M = &sinefcn_M_;

/* Model step function */
void sinefcn_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Trigonometry: '<Root>/Trigonometric Function'
   */
  sinefcn_Y.Out1 = sin_dbl(sinefcn_U.In1);
}

/* Model initialize function */
void sinefcn_initialize(void)
{
  /* Registration code */
```

Inspect the generated code and see if the function or operator replacement occurred as you expected.

Note If a function or operator replacement did not happen as you expected, it means that a call site request was not matched as you intended by your table entry attributes. Either a higher-priority (lower priority value) match was used or no match was found. You can analyze the TFL table entry matching behavior by using the following resources together:

- TFL Viewer, as described in “Using the TFL Viewer to Examine Your Table” on page 6-59 and “Using the TFL Viewer to Examine Registered TFLs” on page 6-60
- HTML code generation reports, with bidirectional tracing including block-to-code highlighting as described above
- Statistics for TFL cache hits and misses logged during code generation, as described in “Examining TFL Cache Hits and Misses” on page 6-64

Examining TFL Cache Hits and Misses

Target function library (TFL) replacement may behave differently than you expect in some cases. To verify that you are obtaining the function or operator replacement that you expect, you first inspect the generated code, as described in "Tracing Code Generated Using Your TFL" on page 6-62.

To analyze replacement behavior, in addition to referencing the generated code and examining your TFL tables in the TFL Viewer, you can view the TFL cache hits and misses logged during the most recent code generation session. This approach provides information on what data types and attributes should be registered in order to achieve the desired replacement.

To display the TFL cache hits and misses logged during the most recent code generation session, you specify the model parameter `TargetFcnLibHandle` in a `get_param` call, as follows:

```
tfl=get_param('model', 'TargetFcnLibHandle')
```

The resulting display includes the following fields:

Field	Description
`HitCache`	Table containing function entries that were successfully matched during a code generation session. These entries represent function implementations that should appear in the generated code.
`MissCache`	Table containing function entries that failed to match during a code generation session. These entries are created by the code generation process for the purpose of querying the TFL to locate a registered implementation. If there is a registered implementation that you feel should have been used in the generated code and was not, examining the `MissCache` for entries that are similar but did not match can help you locate discrepancies in a conceptual argument list or in table entry attributes.

In the following example, the most recent code generation session logged one cache hit and zero cache misses. The logged `HitCache` entry is examined using its table index.

```
>> a=get_param('sinefcn','TargetFcnLibHandle')

a =

RTW.TflControl
        Version: '1.0'
       HitCache: [1x1 RTW.TflCFunctionEntry]
      MissCache: [0x1 handle]
    TLCCallList: [0x1 handle]
      TflTables: [2x1 RTW.TflTable]

>> a.HitCache(1)

ans =

RTW.TflCFunctionEntry
                        Key: 'sin'
                   Priority: 100
             ConceptualArgs: [2x1 RTW.TflArgNumeric]
             Implementation: [1x1 RTW.CImplementation]
          RTWmakecfgLibName: ''
                GenCallback: ''
                GenFileName: ''
             SaturationMode: 'RTW_SATURATE_UNSPECIFIED'
               RoundingMode: 'RTW_ROUND_UNSPECIFIED'
            AcceptExprInput: 1
                SideEffects: 0
                 UsageCount: 2
           SharedUsageCount: 0
                Description: ''
                   ImplType: 'FCN_IMPL_FUNCT'
      AdditionalHeaderFiles: {0x1 cell}
     AdditionalIncludePaths: {0x1 cell}
      AdditionalSourceFiles: {0x1 cell}
      AdditionalSourcePaths: {0x1 cell}
         AdditionalLinkObjs: {0x1 cell}
    AdditionalLinkObjsPaths: {0x1 cell}

>>
```

Registering Target Function Libraries

In this section...

Overview

After you have defined function and operator replacements in a target function library (TFL) table definition file, your table can be included in a TFL that you register with Simulink. Once registered, the TFL will appear in the **Target function library** drop-down list in Simulink, and can be selected for use in model builds.

To register TFLs, use the Simulink customization file `sl_customization.m`. This file is a mechanism that allows you to use M-code to perform customizations of the standard Simulink user interface. Simulink reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see "Customizing the Simulink User Interface" in the Simulink documentation.

Using the sl_customization API to Register a TFL

To register a TFL, you create an instance of `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function accepts one argument: a handle to an object called the `Simulink.CustomizationManager`. The function is declared as follows:

```
function sl_customization(cm)
```

The body of the `sl_customization` function invokes the `registerTargetInfo(tfl)` method provided by `Simulink.CustomizationManager` to register one or more TFLs with Simulink. Typically the`registerTargetInfo` function call references a local function that defines the TFLs to be registered. For example,

```
    % Register the TFL defined in local function locTflRegFcn
    cm.registerTargetInfo(@locTflRegFcn);

  end % End of SL_CUSTOMIZATION
```

Below the `sl_customization` function, the referenced local function describes one or more TFLs to be registered. For example, the local function might be declared as follows:

```
  % Local function to define a TFL
  function thisTfl = locTflRegFcn
```

In the local function body, for each TFL to be registered, you instantiate a TFL registry entry using `tfl = RTW.TflRegistry`. For example,

```
    thisTfl = RTW.TflRegistry;
```

Then you define the following TFL properties within the registry entry:

TFL Property	Description
`Name`	String specifying the name of the TFL, as it will be displayed in the Simulink **Target function library** drop-down list.
`Description`	String specifying a text description of the TFL, which will be displayed as part of the Simulink tool tip for the TFL.
`TableList`	Cell array of strings specifying the tables that make up the TFL, in descending priority order.
`BaseTfl`	String specifying the name of the base TFL on which this TFL is based. To help ensure compatibility between releases, you must specify one of the default MathWorks libraries: `'C89/C90 (ANSI)'`, `'C99 (ISO)'`, `'GNU99 (GNU)'`, or an equivalent alias.
`TargetHWDeviceType`	Always specify `{'*'}`.

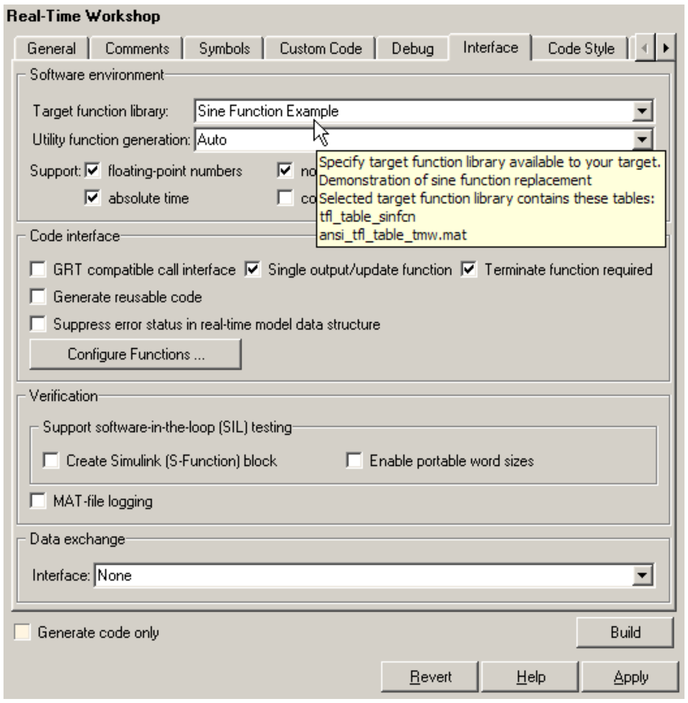
For example:

```
  thisTfl.Name = 'Sine Function Example';
  thisTfl.Description = 'Demonstration of sine function replacement';
  thisTfl.TableList = {'tfl_table_sinfcn'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

Combining the elements described in this section, the complete `sl_customization` function for the `'Sine Function Example'` TFL would appear as follows:

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION


% Local function to define a TFL containing tfl_table_sinfcn
function thisTfl = locTflRegFcn

  % Instance a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Sine Function Example';
  thisTfl.Description = 'Demonstration of sine function replacement';
  thisTfl.TableList = {'tfl_table_sinfcn'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

If you place the `sl_customization.m` file containing this function in the MATLAB search path or in the current working directory, the TFL will be registered at each Simulink startup. Simulink will display the TFL in the **Interface** pane of the Configuration Parameters dialog box. For example, here is the Simulink display, including tool tip, for the `'Sine Function Example'` TFL.



Real-Time Workshop

Software environment

Target function library: Sine Function Example

Utility function generation: Auto

Support: ☑ floating-point numbers ☑ no… ☑ absolute time ☐ co…

Specify target function library available to your target.
Demonstration of sine function replacement
Selected target function library contains these tables:
tfl_table_sinfcn
ansi_tfl_table_tmw.mat

Code interface

☐ GRT compatible call interface ☑ Single output/update function ☑ Terminate function required

☐ Generate reusable code

☐ Suppress error status in real-time model data structure

Verification

Support software-in-the-loop (SIL) testing

☐ Create Simulink (S-Function) block ☐ Enable portable word sizes

☐ MAT-file logging

Data exchange

Interface: None

☐ Generate code only

Tip

- To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`.
- To list all `sl_customization` files in the current search path, use the command `which sl_customization -all`.
- If you disable a TFL registration (for example, by renaming the registration file `sl_customization.m` and then issuing `sl_refresh_customizations`), you may want to reset and save the **Target function library** option setting in any saved models that selected the disabled TFL.

Registering Multiple TFLs

For an example of an `sl_customization` function that registers multiple TFLs, see the `sl_customization.m` file used in the TFL demo, `rtwdemo_tfl_script`. The following excerpt illustrates the general approach.

```
function sl_customization(cm)

  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION

% Local function(s)
function thisTfl = locTflRegFcn
  % Register a Target Function Library for use with model: rtwdemo_tfladdsub.mdl
  thisTfl(1) = RTW.TflRegistry;
  thisTfl(1).Name = 'Addition & Subtraction Examples';
  thisTfl(1).Description = 'Demonstration of addition/subtraction operator replacement';
  thisTfl(1).TableList = {'tfl_table_addsub'};
  thisTfl(1).BaseTfl = 'C89/C90 (ANSI)';
  thisTfl(1).TargetHWDeviceType = {'*'};
  .
  .
  .
  % Register a Target Function Library for use with model: rtwdemo_tflmath.mdl
  thisTfl(4) = RTW.TflRegistry;
  thisTfl(4).Name = 'Math Function Examples';
  thisTfl(4).Description = 'Demonstration of math function replacement';
  thisTfl(4).TableList = {'tfl_table_math'};
  thisTfl(4).BaseTfl = 'C89/C90 (ANSI)';
  thisTfl(4).TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

Target Function Library Limitations

Target function library (TFL) replacement may behave differently than you expect in some cases. For example, data types that you observe in a model do not necessarily match what the code generator determines to use as intermediate data types in an operation. To verify whether you are obtaining the function or operator replacement that you expect, you can inspect the generated code.

To analyze replacement behavior, in addition to referencing the generated code and examining your TFL tables in the TFL Viewer, you can view the TFL cache hits and misses logged during the most recent code generation session. This approach provides information on what data types should be registered in order to achieve the desired replacement. For more information on analyzing TFL table entries, see “Examining and Validating Function Replacement Tables” on page 6-58.

ERT Target Requirements, Restrictions, and Control Files

Requirements and Restrictions for ERT-Based Simulink Models

- For code generation with Real-Time Workshop Embedded Coder, configure your model for the following options on the **Solver** pane of the Configuration Parameters dialog box:
  - **Type**: `fixed-step`
  - **Solver**: You can select any available solver algorithm.
  - **Tasking mode for periodic sample times**: When the model is single-rate, you must select the `SingleTasking` or `Auto` mode. Permitted Solver Modes for Real-Time Workshop Embedded Coder Targeted Models on page 1-15 indicates permitted solver modes for single-rate and multirate models.
- If you use blocks that have a dependency on absolute time in a program, you should properly specify the **Application lifespan (days)** parameter on the **Optimization** pane. (See "Blocks That Depend on Absolute Time" in the Real-Time Workshop documentation for a list of such blocks.) You can use these blocks in applications that run for extremely long periods, with counters that provide accurate and overflow-free absolute time values, provided that you specify a long enough lifespan. If you are designing a program that is intended to run indefinitely, specify **Application lifespan (days)** as `inf`. This generates a 64 bit integer counter. For an application whose sample rate is 1000 MHz, a 64 bit counter will not overflow for more than 500 years.
- You can use any Simulink blocks in your models, except for blocks not supported by the `Embedded-C` format, as follows:
  - MATLAB Fcn
  - M-file and Fortran S-functions that are not inlined with TLC

  Note that use of certain blocks is not recommended for production code generation for embedded systems. To view a table that summarizes characteristics of blocks in the Simulink and Fixed-Point block libraries, execute the following command at the MATLAB command line:

  ```
  showblockdatatypetable
  ```

Refer to the **Code Generation Support** column of the table and its footnotes, including the footnote "Not recommended for production code."

- You can use both inlined and non-inlined S-functions with Real-Time Workshop Embedded Coder. However, inlined S-functions are often advantageous in production code generation, for example in implementing device drivers. See "Tradeoffs in Device Driver Development" in the Developing Embedded Targets document for a discussion of the pros and cons.

ERT System Target File and Template Makefiles

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

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

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

Examples

Use this list to find examples in the documentation.

Data Structures, Code Modules, and Program Execution

Index